Using a theorem prover for reasoning on constraint problems

Marco Cadoli and Toni Mancini

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

Abstract. Specifications of constraint problems can be considered logical formulae. As a consequence, it is possible to infer their properties by means of automated reasoning tools. The purpose of this paper is exactly to link two important technologies: automated theorem proving and constraint programming. We report the results on using a theorem prover and a finite model finder for checking existence of symmetries, checking whether a given formula breaks a symmetry, and checking existence of functional dependencies among groups of predicates. As a side-effect, we propose a new domain of application and a brand new set of benchmarks for ATP systems.

1 Introduction

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. In fact, many systems and languages for the solution of constraint problems (e.g., AMPL [9], OPL [20], GAMS [5], DLV [7], SMODELS [18], and NP-SPEC [4]) clearly separate the *specification* of a problem, e.g., graph 3-coloring, and its *instance*, e.g., *a* graph, using a two-level architecture for finding solutions: the specification is instantiated (or grounded) against the instance, and then an appropriate solver is invoked. There are several benefits in this separation: obviously declarativeness increases, and the solver is completely decoupled from the specification. Ideally, the programmer can focus only on the specification of the problem, without committing *a priori* to a specific solver. In fact, some systems, e.g., AMPL [9], are able to translate –at the request of the user– a specification in various formats, suitable for different solvers.

Again, the syntax varies a lot among such languages: AMPL, OPL, 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) on finite databases, where the existential second-order quantifiers and the first-order formula represent, respectively, the *guess* and *check* phases of the

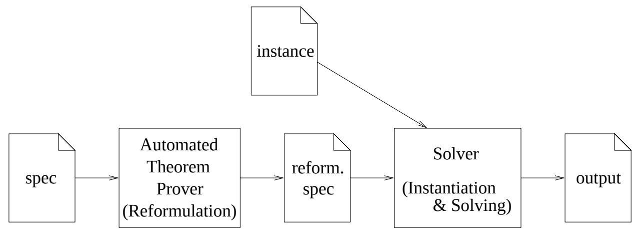

Fig. 1. Architecture of the problem solving system.

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.

Since specifications are logical formulae, it is possible to infer their properties by means of automated reasoning tools. The purpose of this paper is exactly to link two important technologies: automated theorem proving (ATP) and constraint programming. The architecture of the system we envision is represented in Figure 1.

In particular, we report the results on using a theorem prover and a finite model finder for reasoning on specifications of constraint problems, represented as ESO formulae. We focus on two forms of reasoning:

– checking existence of *value symmetries*, i.e., properties of the specification that allow to exchange values of the finite domains without losing all solutions; on top of that, we check whether a given formula breaks such symmetries;
– checking existence of *functional dependencies*, i.e., properties that force values of some guessed predicates to depend on the value of some others.

There are at least two reasons why a system should make automatically such checks: first of all, it has been proven that solving can be made much more efficient by, e.g., recognizing and breaking symmetries (a wide literature is nowadays available, cf., e.g., [2, 6]). Secondly, the person performing constraint modelling may be interested in the above properties: as an example, existence (or lack) of a dependency may reveal a bug in the specification.

The main result of this paper is that it is actually possible to use ATP technology to reason on combinatorial problems, and we exhibit several examples proving it. As a side-effect, we propose a new domain of application and a brand new set of benchmarks for ATP systems, which is not represented in large repositories, such as TPTP (cf. `www.tptp.org`).

Relations between constraint satisfaction and deduction have been observed since several years (cf., e.g., the early work [1], and [12] for an up-to-date report). The use of automated tools for preprocessing constraint satisfaction problems (CSPs) has been limited, to the best of our knowledge, to the *instance* level. As an example, the use of packages such as *nauty* [16] for finding symmetries on CSPs has been proposed in [6]. On the other hand, not much work has been done on reasoning at the *specification* level. A limited form of reasoning is offered by the OPL system, which checks (syntactically) whether a specification contains only linear constraints and objective function, and in this case invokes an integer linear programming solver (typically very efficient); otherwise, it uses a constraint programming solver.

The rest of the paper is organized as follows: in Section 2 we give some preliminaries on modelling combinatorial problems as formulae in ESO. Sections 3 and 4 are devoted to the description of experiments in checking symmetries and dependencies, respectively. In Section 5 we conclude the paper, and present current research.

2 Preliminaries

In this paper, we use *existential second-order logic* (ESO) for the specification of problems, which allows to represent all search problems in the complexity class NP [8]. The use of ESO as a modelling language for problem specifications is common in the database literature, but unusual in constraint programming, therefore few comments are in order. Constraint modelling systems like those mentioned in Section 1 have a richer syntax and more complex constructs, and we plan to eventually move from ESO to such languages. For the moment, we claim that studying the simplified scenario is a mandatory starting point for more complex investigations, and that our results can serve as a basis for reformulating specifications written in higher-level languages. Anyway, examples using the syntax of the implemented language OPL are exhibited in Sections 4 and 5.

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 ψ_π

$$\exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R}), \tag{1}$$

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

An instance $\mathcal{I}$ to the problem is given as a relational database over the schema $\boldsymbol{R}$, i.e., as an extension for all relations in $\boldsymbol{R}$. Predicates (of given arities) in the set $\boldsymbol{S} = \{S_1, \ldots, S_n\}$ 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.

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

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

It is worthwhile to note that, when a specification is instantiated against an input database, a CSP is obtained.

Example 1 (Graph 3-coloring). In this NP-complete decision problem (cf. [10, Prob. GT4, p. 191]) the input is a graph, and the question is whether it is possible to give each of its nodes one out of three colors (red, green, and blue), in such a way that adjacent nodes (not including self-loops) are never colored the same way. The question can be easily specified as an ESO formula ψ on the input schema $\boldsymbol{R} = \{edge(\cdot,\cdot)\}$:

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

3 Value symmetries

In this section we face the problem of automatically detecting and breaking some symmetries in problem specifications. In Subsection 3.1 we give preliminary definitions of problem transformation and symmetry taken from [3], and show how the symmetry-detection problem can be reduced to checking semantic properties of first-order formulae. We limit our attention to specifications with monadic guessed predicates only, and to transformations and symmetries on values. Motivations for these limitations are given in [3]; here, we just recall that non-monadic guessed predicates can be transformed in monadic ones by unfolding and by exploiting the finiteness of the input database. We refer to [3] also for considerations on benefits of the technique on the efficiency of problem solving. In Subsection 3.2 we then show how a theorem prover can be used to automatically detect and break symmetries.

3.1 Definitions

Definition 1 (Uniform value transformation (UVT) of a specification [3]). *Given a problem specification $\psi \doteq \exists \boldsymbol{S}\ \phi(\boldsymbol{S},\boldsymbol{R})$, with $\boldsymbol{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1,n]$, and input schema $\boldsymbol{R}$, a* uniform value transformation (UVT) *for ψ is a mapping $\sigma : \boldsymbol{S} \rightarrow \boldsymbol{S}$, which is total and onto, i.e., defines a permutation of guessed predicates in $\boldsymbol{S}$.*

The term “uniform value” transformation in Definition 1 is used because swapping monadic guessed predicates is conceptually the same as uniformly exchanging domain values in a CSP.

From here on, given ϕ and σ as in the above definition, ϕ^σ is defined as $\phi[S_1/\sigma(S_1), \ldots, S_n/\sigma(S_n)]$, i.e., ϕ^σ is obtained from ϕ by uniformly substituting every occurrence of each guessed predicate with the one given by the transformation σ. Analogously, ψ^{σ} is defined as $\exists \boldsymbol{S}\ \phi^{\sigma}(\boldsymbol{S}, \boldsymbol{R})$.

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

Definition 2 (Uniform value symmetry (UVS) of a specification [3]). *Let $\psi \doteq \exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R})$, be a specification, with $\boldsymbol{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1, n]$, and input schema $\boldsymbol{R}$, and let σ be a UVT for ψ. Transformation σ is a* uniform value symmetry (UVS) *for ψ if the following holds:*

$$\forall \boldsymbol{S}\ \forall \mathcal{I}\ \ \phi(\boldsymbol{S}, \mathcal{I})\ \leftrightarrow\ \phi^{\sigma}(\boldsymbol{S}, \mathcal{I}). \tag{9}$$

Intuitively, formula (9) says that σ is a UVS for ψ if every extension for $\boldsymbol{S}$ which satisfies ϕ, satisfies also ϕ^{σ} and vice versa, for every input instance $\mathcal{I}$, i.e., for every extension of the input schema $\boldsymbol{R}$. Note that every CSP obtained by instantiating a specification with σ has at least the corresponding uniform value symmetry.

In [3], it is shown that checking whether a UVT is a UVS reduces to checking equivalence of two first-order formulae:

Proposition 1 ([3]). *Let ψ be a problem specification of the kind* (1)*, with only monadic guessed predicates, and σ a UVT for ψ. Transformation σ is a UVS for ψ if and only if $\phi \equiv \phi^{\sigma}$.*

Once symmetries of a specification have been detected, additional constraints can be added in order to *break* them, i.e., to wipe out from the solution space (some of) the symmetrical points. These kind of constraints are called *symmetry-breaking formulae*, and are defined as follows.

Definition 3 (Symmetry-breaking formula [3]). *Let $\psi \doteq \exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R})$, be a specification, with $\boldsymbol{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1, n]$, and input schema $\boldsymbol{R}$, and let σ be a UVS for ψ. A* symmetry-breaking formula *for ψ with respect to symmetry σ is a closed (except for $\boldsymbol{S}$) formula $\beta(\boldsymbol{S})$ such that the following two conditions hold:*

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

$$(\phi \wedge \beta(\boldsymbol{S})) \not\equiv (\phi \wedge \beta(\boldsymbol{S}))^{\sigma}\ ;$$

2. *Every model of ϕ can be obtained by those of $\phi \wedge \beta(\boldsymbol{S})$ and their symmetric with respect to σ:*

$$\phi\ \equiv\ (\phi \wedge \beta(\boldsymbol{S}))\ \vee\ (\phi \wedge \beta(\boldsymbol{S}))^{\sigma}\ .$$

If $\beta(\boldsymbol{S})$ matches the above definition, then we are entitled to solve the problem $\exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R}) \wedge \beta(\boldsymbol{S})$ instead of the original one $\exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R})$. In fact, point 1 in the above definition states that formula $\beta(\boldsymbol{S})$ actually breaks σ, since, by Proposition 1, σ is not a symmetry of the rewritten problem. Point 2 guarantees that all solutions are preserved in the rewritten problem, up to symmetric ones, which can be readily obtained by applying σ.

3.2 Experiments with the theorem prover

Proposition 1 suggests that the problem of detecting UVSs of a specification ψ of the kind (1) can in principle be performed in the following way:

1. Selecting a UVT σ, i.e. a permutation of guessed predicates in ψ (if ψ has n guessed predicates, there are $n!$ such UVTs);
2. Checking whether σ is a UVS, i.e., deciding whether $\phi \equiv \phi^{\sigma}$.

The above procedure suggests that a first-order theorem prover can be used to perform automatically point 2. Even if we proved in [3] that this problem is undecidable, we show how a theorem prover usually performs well on this kind of formulae.

As for the symmetry-breaking problem, from conditions of Definition 3 it follows that also the problem of checking whether a formula breaks a given UVS for a specification clearly reduces to semantic properties of logic formulae.

In this section we give some details about the experimentation done using automated tools. First of all we note that, obviously, all the above conditions can be checked by using a refutation theorem prover. It is interesting to note that, for some of them, we can use a finite model finder. In particular, we can use such a tool for checking statements (such as condition 1 of Definition 3 or the negation of the condition of Proposition 1) which are syntactically a non-equivalence. As a matter of facts, it is enough to look for a finite model of the negation of the statement, i.e., the equivalence. If we find such a model, then we are sure that the non-equivalence holds, and we are done. The tools we used are OTTER [15], and MACE [14], respectively.

Detecting symmetries The examples on which we worked are the following.

Example 2 (Graph 3-coloring: Example 1 continued). The mapping $\sigma^{R,G} : \boldsymbol{S} \rightarrow \boldsymbol{S}$ such that $\sigma^{R,G}(R) = G$, $\sigma^{R,G}(G) = R$, $\sigma^{R,G}(B) = B$ is a UVT for it. It is easy to observe that formula $\phi^{\sigma^{R,G}}$ is equivalent to ϕ, because clauses of the former are syntactically equivalent to clauses of the latter and vice versa. This implies, by Proposition 1, that $\sigma^{R,G}$ is also a UVS for the specification of the 3-coloring problem. The same happens also for transformations $\sigma^{R,B}$ and $\sigma^{G,B}$ that swap B with, respectively, R and G.

Example 3 (Not-all-equal Sat). In this NP-complete problem [10], the input is a propositional formula in CNF, and the question is whether it is possible to assign a truth value to all the variables in such a way that the input formula is satisfied, and that every clause contains at least one literal which truth value is false. We assume that the input formula is encoded by the following relations:

- $inclause(\cdot,\cdot)$; tuple $\langle l, c\rangle$ is in $inclause$ iff literal l is in clause c;
- $l^{+}(\cdot,\cdot)$; a tuple $\langle l, v\rangle$ is in l^{+} iff l is the positive literal relative to the propositional variable v, i.e., v itself;
- $l^{-}(\cdot,\cdot)$; a tuple $\langle l, v\rangle$ is in l^{-} iff l is the negative literal relative to the propositional variable v, i.e., $\neg v$;

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

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

$$\exists TF\ \forall X\ var(X) \leftrightarrow T(X) \vee F(X)\ \ \wedge \tag{10}$$
$$\forall X\ \neg\,(T(X) \wedge F(X))\ \ \wedge \tag{11}$$
$$\forall C\ clause(C) \rightarrow$$
$$\big[\exists L\ inclause(L{,}C) \wedge \forall V\ \big(l^{+}(L{,}V){\rightarrow}T(V)\big) \wedge \big(l^{-}(L{,}V){\rightarrow}F(V)\big)\,\big]\ \wedge \tag{12}$$
$$\forall C\ clause(C) \rightarrow$$
$$\big[\exists L\ inclause(L{,}C) \wedge \forall V\ \big(l^{+}(L{,}V){\rightarrow}F(V)\big) \wedge \big(l^{-}(L{,}V){\rightarrow}T(V)\big)\,\big]. \tag{13}$$

Constraints (10–11) force every variable to be assigned exactly one truth value; moreover, (12) forces the assignment to be a model of the formula, while (13) leaves in every clause at least one literal whose truth value is false.

Let us consider the UVT $\sigma^{T,F}$, defined as $\sigma^{T,F}(T) = F$ and $\sigma^{T,F}(F) = T$. It is easy to prove that $\sigma^{T,F}$ is a UVS for this problem, since $\phi^{\sigma^{T,F}}$ is equivalent to ϕ.

The results we obtained with OTTER are shown in Table 1. The third row refers to the version of the Not-all-equal Sat problem in which all clauses have three literals, the input is encoded using a ternary relation $clause(\cdot,\cdot,\cdot)$, and the specification varies accordingly. It is interesting to see that the performance is always quite good.

A note on the encoding is in order. Initially, we gave the input to OTTER exactly in the format specified by Proposition 1, but the performance was quite poor: for 3-coloring the tool did not even succeed in transforming the formula in clausal form, and symmetry was proven only for very simplified versions of the problem, e.g., 2-coloring, omitting constraint (2). Results of Table 1 have been obtained by introducing new propositional variables defining single constraints. as an example, constraint (2) is represented as

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

where `covRGB` is a fresh propositional variable. Obviously, we wrote a first-order logic formula encoding condition of Proposition 1, and gave its negation to OTTER in order to find a refutation.

As for proving non-existence of symmetries, we used the following example.

Example 4 (Graph 3-coloring with red self-loops). We consider a modification of the problem of Example 1, and show that only one of the UVTs in Example 2 is indeed a UVS for the new problem. Here, the question is whether it is possible to 3-color the input graph in such a way that every self loop insists on a red node.

Spec	Symmetry	CPU time (sec)	Proof length	Proof level
3-coloring	$\sigma^{R,G}$	0.27	43	12
Not-all-equal Sat	$\sigma^{T,F}$	0.22	54	19
Not-all-equal 3-Sat	$\sigma^{T,F}$	4.71	676	182

Table 1. Performance of OTTER for proving that a UVT is a UVS.

In ESO, one more clause (which forces the nodes with self loops to be colored in red) must be added to the specification in Example 1:

$$\forall X \quad edge(X,X) \rightarrow R(X). \tag{14}$$

UVT $\sigma^{G,B}$ is a UVS also of the new problem, because of the same argument of Example 2. However, for what concerns $\sigma^{R,G}$, in this case $\phi^{\sigma^{R,G}}$ is not equivalent to ϕ: as an example, for the input instance $edge = \{(v,v)\}$, the color assignment $\overline{R}, \overline{G}, \overline{B}$ such that $\overline{R} = \{v\}, \overline{G} = \overline{B} = \emptyset$ is a model for the original problem, i.e., $\overline{R}, \overline{G}, \overline{B} \models \phi(R,G,B,edge)$. It is however easy to observe that $\overline{R}, \overline{G}, \overline{B} \not\models \phi^{\sigma^{R,G}}(R,G,B,\ edge)$, because $\phi^{\sigma^{R,G}}$ is verified only by color assignments for which $\overline{G}(v)$ holds. This implies, by Proposition 1, that $\sigma^{R,G}$ is not a UVS. For the same reason, also $\sigma^{R,B}$ is not a UVS for the new problem.

We wrote a first-order logic formula encoding condition of Proposition 1 for $\sigma^{R,G}$ on the above example and gave its negation to MACE in order to find a model of the non-equivalence. MACE was able to find the model described in Example 4 in less than one second of CPU time.

Breaking symmetries We worked on the 3-coloring problem specification given in Example 1 and the UVS $\sigma^{R,G}$ defined in Example 2. This UVS can be broken in several ways, as an example by the following formula:

$$\beta^{R,G}_{sel}(R,G,B) \doteq R(\overline{v}) \vee B(\overline{v}), \tag{15}$$

that forces a selected node, say $\overline{v}$, not to be colored in green. The simpler formula $R(\overline{v})$ breaks two symmetries, namely $\sigma^{R,G}$ and $\sigma^{R,B}$, and can be obtained as the logical and of (15) and $R(\overline{v}) \vee G(\overline{v})$.

We used MACE and OTTER in order to prove that (15) is indeed a symmetry-breaking formula for the 3-coloring problem specification with respect to $\sigma^{R,G}$, i.e., for testing conditions 1 and 2 (respectively) of Definition 3. Both systems succeeded in less than one second of CPU time.

As described in [3], a UVS can be broken in several ways, and with different effectiveness. As an example, $\sigma^{R,G}$ in the 3-coloring problem specification can be broken also by the following formula:

$$\beta^{R,G}_{card}(R,G,B) \ \doteq \ |R| \leq |G|, \tag{16}$$

that forces green nodes to be at least as many as red ones. It is easy to prove that formula (16) respects both conditions of Definition 3, and of course is much more effective than formula (15) to break the symmetry.

However, this example highlights some difficulties that can arise when using first-order ATPs. In fact, although constraint (16) can be written in ESO using standard techniques, it is not first-order definable. Therefore, conditions in Definition 3 are (non-)equivalence of second-order formulae. So, the use of a first-order theorem prover may in general not suffice.

We are currently investigating whether it is possible to define first-order conditions that can be used to infer the truth value of those of Definition 3 in such cases.

4 Dependent predicates

In this section we tackle the problem of recognizing guessed predicates that functionally depend on the others in a given specification. This means that, for every solution of any instance, the extension of a dependent guessed predicate is determined by the extensions of the others.

Recognizing functionally dependent predicates in a specification is very important for the efficiency of any backtracking solver, since branches regarding dependent predicates (that represent values assigned to variables of the CSP obtained after instantiation) can be safely avoided. As an example, it is shown in [11] how to modify the Davis-Putnam procedure for SAT so that is avoids branches on variables added during the clausification of non-CNF formulae, since values assigned to these variables depend on assignments to the other ones. Moreover, specific SAT solvers, e.g., EQSATZ [13], have been developed in order to appropriately handle (by means of the so-called "equivalence reasoning") equivalence clauses, which have been recognized to be a very common structure in the SAT encoding of many hard real-world problems, and a major obstacle to the Davis-Putnam procedure.

The automatic recognition of functionally dependent predicates is important also to improve the quality aspects of specifications as software artefacts. A dependent predicate may be an evidence either of a bug in the problem model, or of a bad design choice, since the adopted model for the problem is, in some sense, redundant. Of course, the use of dependent predicates can also be the consequence of a precise design choice, e.g., with the goal of a more modular and readable problem specification. In any case, a feature of the system that automatically checks whether a dependence holds is likely to be useful to the designer.

In Subsection 4.1, we give the formal definition of dependent predicates in a specification, and in Subsection 4.2 show how the problem of checking whether a set of guessed predicates is dependent from the others reduces to check semantic properties of a first-order formula. We observe that definitions and results in this section are original, and do not appear elsewhere. Proof of theorems are not given for space reasons, and will appear in the full paper.

4.1 Definitions

Definition 4 (Functional dependence of a set of predicates in a specification). *Given a problem specification $\psi \doteq \exists \boldsymbol{S}\boldsymbol{P}\ \phi(\boldsymbol{S},\boldsymbol{P},\boldsymbol{R})$, with input schema $\boldsymbol{R}$, $\boldsymbol{P}$* functionally depends *on $\boldsymbol{S}$ if, for each instance $\mathcal{I}$ of $\boldsymbol{R}$ and for each pair of interpretations M, N of $(\boldsymbol{S},\boldsymbol{P})$ it holds that, if*

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

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

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

Example 5 (Graph 3-coloring: Example 1 continued). In the 3-coloring problem, one of the three guessed predicates is functionally dependent on the others. As an example, B functionally depends on R and G, since, regardless of the instance, it can be defined as $\forall X\ B(X) \leftrightarrow \neg(R(X) \vee G(X))$: constraint (2) is equivalent to $\forall X\ \neg(R(X) \vee G(X)) \rightarrow B(X)$ and (4) and (5) imply $\forall X\ B(X) \rightarrow \neg(R(X) \vee G(X))$. In other words, for every input instance, no two different solutions exist that coincide on the set of red and green nodes, but differ on the set of blue ones.

Example 6 (Not-all-equal Sat: Example 3 continued). One of the two guessed predicates T and F is functionally dependent on the other, since by constraints (10–11) it follows, e.g., $\forall X\ F(X) \leftrightarrow var(X) \wedge \neg T(X)$.

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

$$\bigwedge_{T \in \boldsymbol{T}} \forall \boldsymbol{X}\ T(\boldsymbol{X}) \equiv T'(\boldsymbol{X}),$$

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

Theorem 1. *Let $\psi \doteq \exists \boldsymbol{S}\boldsymbol{P}\ \phi(\boldsymbol{S},\boldsymbol{P},\boldsymbol{R})$ be a problem specification with input schema $\boldsymbol{R}$. $\boldsymbol{P}$ functionally depends on $\boldsymbol{S}$ if and only if the following formula is valid:*

$$[\phi(\boldsymbol{S},\boldsymbol{P},\boldsymbol{R}) \wedge \phi(\boldsymbol{S'},\boldsymbol{P'},\boldsymbol{R}) \wedge \neg(\boldsymbol{S}\boldsymbol{P} \equiv \boldsymbol{S'}\boldsymbol{P'})] \rightarrow \neg(\boldsymbol{S} \equiv \boldsymbol{S'}). \tag{17}$$

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

Theorem 2. *Given a specification on input schema $\boldsymbol{R}$, and a partition $(\boldsymbol{S}, \boldsymbol{P})$ of its guessed predicates, the problem of checking whether $\boldsymbol{P}$ functionally depends on $\boldsymbol{S}$ is not decidable.*

Nonetheless, as shown in the next section, an ATP usually performs very well in deciding whether formulae of the kind of (17) are valid or not.

4.2 Experiments with the theorem prover

Using Theorem 1 it is easy to write a first-order formula that is valid if and only if a given dependency holds. We used OTTER for proving the existence of dependencies among guessed predicates of different problem specifications:

- Graph 3-coloring (cf. Example 5), where one among the guessed predicates R, G, B is dependent on the others.
- Not-all-equal Sat (cf. Example 6), where one between the guessed predicates T and F is dependent on the other.

For each of the above specifications, we wrote a first-order logic encoding of formula (17), and gave its negation to OTTER in order to find a refutation.

For the purpose of testing effectiveness of the proposed technique in the context of specifications written in implemented languages, we considered also the *Sailco inventory* problem, taken from the OPL book [20, Section 9.4, Statement 9.17] and part of the OPLSTUDIO distribution package (as file `sailco.mod`).

Example 7 (The Sailco inventory problem). This problem specification models a simple inventory application, in which the question is to decide how many sailboats the Sailco company has to produce over a given number of time periods, in order to satisfy the demand and to minimize production costs. The demand for the periods is known and, in addition, an `inventory` of boats is available initially. In each period, Sailco can produce a maximum number of boats (`capacity`) at a given unitary cost (`regularCost`). Additional boats can be produced, but at higher cost (`extraCost`). Storing boats in the inventory also has a cost per period (`inventoryCost` per boat).

Figure 2 shows an OPL model for this problem. An equivalent –apart, of course, for the objective function– ESO specification would be more complex, because of the presence of arithmetic operations in the constraints, and thus will not be presented. However, the analogous of the instance relational schema, guessed predicates, and constraints can be clearly distinguished in the OPL code. Guessed predicates of the ESO specification can be obtained in standard ways: as an example, for the inventory we can define a guessed predicate $inv(\cdot, \cdot)$ with the first argument being the period, and the second one the amount of boats stored in that period, plus additional constraints to force exactly one tuple to belong to $inv(\cdot, \cdot)$ for each period.

```
// Instance schema
int+ nbPeriods = ...;            range Periods 1..nbPeriods;

float+ demand[Periods] = ...;    float+ regularCost = ...;
float+ extraCost = ...;          float+ capacity = ...;
float+ inventory = ...;          float+ inventoryCost = ...;

// Guessed predicates
var float+ regulBoat[Periods];   var float+ extraBoat[Periods];
var float+ inv[0..nbPeriods];

// Objective function
minimize ...

// Constraints
subject to {
   inv[0] = inventory;
   forall(t in Periods) regulBoat[t] <= capacity;
   forall(t in Periods)
      regulBoat[t] + extraBoat[t] + inv[t-1] = inv[t] + demand[t];
};
```

Fig. 2. OPL specification for the Sailco problem.

From the specification in Figure 2, it can be observed that the amount of boats in the inventory for each time period `t` > 0 (i.e., `inv[t]`) is defined in terms of the amount of regular and extra boats produced in period `t` by the following relationship: `inv[t] = regulBoat[t] + extraBoat[t] - demand[t] + inv[t-1]`. Of course, the same relationship holds in the equivalent ESO specification, making predicate $inv(\cdot,\cdot)$ functionally dependent on $regulBoat(\cdot,\cdot)$ and $extraBoat(\cdot,\cdot)$.

We opted for an OTTER encoding that uses function symbols: as an example, the `inv[]` array is translated to a function symbol $inv(\cdot)$ rather than to a binary predicate. More precisely, according to Theorem 1, a pair of function symbols $inv(\cdot)$ and $inv'(\cdot)$ is introduced. The same happens for `regulBoat[]` and `extraBoat[]`. Moreover, we included in the OTTER formula additional constraints in order to make the system able to correctly handle expressions of interest. Notably, the following formulae allow to infer $\forall t\ inv(t) = inv'(t)$ from equality of inv and inv' at the initial time period and equivalence of increments in all time intervals of length 1.

```
equalDiscrete <-> (inv(0) = inv1(0) &
                  (all t (t > 0 -> (inv(t) - inv(t-1)) =
                                  (inv1(t) - inv1(t-1))))).
induction <-> (equalDiscrete -> (all t (inv(t) = inv1(t)))).
```

Spec	$\boldsymbol{S}$	$\boldsymbol{P}$	CPU time (sec)	Proof length	Proof level
3-coloring	R, G	B	0.25	27	18
Not-all-equal 3-Sat	T	F	0.38	18	14
Sailco	$regulBoat$, $extraBoat$	inv	0.21	29	11

Table 2. Performance of OTTER for proving that the set $\boldsymbol{P}$ of guessed predicates is functionally dependent on the set $\boldsymbol{S}$.

Results of the experiments are presented in Table 2. As it can be observed, the time needed by OTTER is always very low.

5 Conclusions and current research

The use of automated tools for preprocessing CSPs has been limited, to the best of our knowledge, to the instance level. In this paper we proved that current ATP technology is able to perform significant forms of reasoning on specifications of constraint problems. We focused on two forms of reasoning: symmetry detection and breaking, and functional dependence checking. Reasoning has been done for various problems, including the ESO encodings of graph 3-coloring and Not-all-equal Sat, and the OPL encoding of an inventory problem. In general, reasoning is done very efficiently by the ATP, although effectiveness depends on the format of the input, and auxiliary propositional variables seem to be necessary.

There are indeed some tasks, namely, proving existence of symmetries in the *Social golfer problem* (problem 10 at `www.csplib.org`) which OTTER –in the automatic mode– was unable to do. So far, we used only two tools, namely OTTER and MACE, and plan to investigate effectiveness of other provers, e.g., VAMPIRE [19]. We note that the wide availability of constraint problem specifications, both in implemented languages, cf., e.g., [9, 20], and in natural language, cf., e.g., [10], the CSP-Library (`www.csplib.org`), the OR-Library (`www.ms.ic.ac.uk/info.html`), offers a brand new set of benchmarks for ATP systems, which is not represented in large repositories, such as TPTP (cf. `www.tptp.org`).

We believe that ATPs can be used also for other useful forms of reasoning, apart from those described in this paper. As an example, in Figure 3 we show the OPL specification of the N-queens problem, cf. [20, Section 2.2, Statement 2.16], which states that three constraints must hold for all pairs of distinct rows (cf. the condition `r1 <> r2`). For symmetry reasons, a solution-preserving (and possibly more efficient) formulation requires the constraints to hold just for *totally ordered* pairs of rows, i.e., `r1 < r2`. From the logical point of view, this can be recognized simply by proving that swapping `r1` and `r2` leads to an equivalent specification. OTTER was able to prove such an equivalence in less than one second of CPU time. We are currently investigating the applicability of such a technique for a general class of specifications, in which symmetries on *variables* [17] hold.

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

Fig. 3. OPL specification for the N-queens problem.

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 indebted to Marco Schaerf for fruitful discussions, and in particular for Definition 4.

References

1. W. Bibel. Constraint satisfaction from a deductive viewpoint. *Artificial Intelligence*, 35:401–413, 1988.
2. C. A. Brown, L. Finkelstein, and P. W. Purdom. Backtrack searching in the presence of symmetry. In T. Mora, editor, *Proc. of 6th Intl. Conf. on Applied Algebra, Algebraic Algorithms and Error Correcting codes*, pages 99–110. Springer, 1988.
3. M. Cadoli and T. Mancini. Detecting and breaking symmetries on specifications. In *Proceedings of The Third International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with CP'03*. Available at `http://scom.hud.ac.uk/scombms/SymCon03/Papers/Cadoli.pdf`, 2003.
4. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. In *Proceedings of the European Symposium On Programming (ESOP 2001)*, volume 2028 of *Lecture Notes in Computer Science*, pages 387–401. Springer, 2001.
5. E. Castillo, A. J. Conejo, P. Pedregal, and N. A. Ricardo Garca. *Building and Solving Mathematical Programming Models in Engineering and Science*. Wiley, 2001.
6. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and 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)*, pages 148–159, 1996.
7. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, Comparisons and Benchmarks. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417, 1998.

8. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In R. M. Karp, editor, *Complexity of Computation*, pages 43–74. AMS, 1974.
9. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson Publishing, 1993.
10. M. R. Garey and D. S. Johnson. *Computers and Intractability—A guide to NP-completeness*. W.H. Freeman and Company, San Francisco, 1979.
11. E. Giunchiglia and 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)*, volume 1792 of *Lecture Notes in Artificial Intelligence*, pages 84–94. Springer, 2000.
12. P. G. Kolaitis. Constraint satisfaction, databases, and logic. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 1587–1595, 2003.
13. C. M. Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *AAAI: 17th National Conference on Artificial Intelligence*. AAAI / MIT Press, 2000.
14. W. McCune. Mace 2.0 reference manual and guide. Technical Report ANL/MCS-TM-249, Argonne National Laboratory, Mathematics and Computer Science Division, May 2001. Available at `http://www-unix.mcs.anl.gov/AR/mace/`.
15. W. McCune. Otter 3.3 reference manual. Technical Report ANL/MCS-TM-263, Argonne National Laboratory, Mathematics and Computer Science Division, August 2003. Available at `http://www-unix.mcs.anl.gov/AR/otter/`.
16. B. D. McKay. *nauty* user's guide (version 2.2). Available at `http://cs.anu.edu.au/~bdm/nauty/nug.pdf`, 2003.
17. P. Meseguer and C. Torras. Solving strategies for highly symmetric CSPs. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 400–405, 1999.
18. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
19. A. Riazanov and A. Voronkov. Vampire. In *Proceedings of the Sixteenth International Conference on Automated Deduction (CADE'99)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, 1999.
20. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.