# An Experimental Comparison of Constraint Logic Programming and Answer Set Programming

**Agostino Dovier**
Università di Udine,
Dip. di Matematica e Informatica
dovier@dimi.uniud.it

**Andrea Formisano**
Università di Perugia
Dip. di Matematica e Informatica
formis@dipmat.unipg.it

**Enrico Pontelli**
New Mexico State University
Dept. of Computer Science
epontell@cs.nmsu.edu

## Abstract

*Answer Set Programming (ASP)* and *Constraint Logic Programming over finite domains (CLP(FD))* are two declarative programming paradigms that have been extensively used to encode applications involving search, optimization, and reasoning (e.g., commonsense reasoning and planning). This paper presents experimental comparisons between the declarative encodings of various computationally hard problems in both frameworks. The objective is to investigate how the solvers in the two domains respond to different problems, highlighting strengths and weaknesses of their implementations, and suggesting criteria for choosing one approach over the other. Ultimately, the work in this paper is expected to lay the foundations for transfer of technology between the two domains, e.g., suggesting ways to use CLP(FD) in the execution of ASP.[1]

## Introduction

The objective of this work is to experimentally compare the use of two distinct logic-based paradigms, traditionally recognized as excellent tools to tackle computationally hard problems and to encode AI applications (such as planning, scheduling, and optimization problems). The two paradigms considered are *Answer Set Programming (ASP)* (Baral 2003) and *Constraint Logic Programming over Finite Domains (CLP(FD))* (Marriott & Stuckey 1998). The motivation for this investigation, originally described in (Dovier, Formisano, & Pontelli 2005), arises from the successful use of both paradigms in dealing with various classes of combinatorial problems, and the need to better understand their respective strengths and weaknesses. Ultimately, we hope this work will indicate methods for integration and cooperation between the two paradigms.

ASP is a paradigm deriving from logic programming under answer set semantics, where problem components are encoded as atomic formulae and relationships as program rules. It is well-known (Marek & Truszczynski 1991; Baral 2003) that, given a propositional normal logic program $P$, deciding whether $P$ admits an *answer set* (Gelfond & Lifschitz 1988) is a NP-complete problem. As a consequence, any NP-complete problem can be encoded as a propositional normal logic program under answer set semantics. Answer-set solvers are programs designed to compute the answer sets of normal logic programs. Some solvers provide also syntactic extensions to facilitate program development—e.g., limited forms of aggregation (Simons 2000)—and classes of *optimization statements*, used to select answer sets that maximize or minimize an objective function dependent on the content of the answer set.

An alternative framework, frequently adopted to handle NP-complete, combinatorial, and optimization problems, is CLP(FD) (Jaffar & Maher 1994; Marriott & Stuckey 1998). In this context, a finite domain of objects (typically integers) is associated to each variable in the problem specification, and the problem-derived relationships between such variables are encoded as constraints (e.g., $s = t$, $s < t$). This type of framework supports the natural and declarative encoding of search strategies and NP-complete problems. Indeed, a rich literature has been developed presenting applications of CLP(FD) to a variety of search and optimization problems (Marriott & Stuckey 1998), and several efficient implementations have been developed.

In this paper, we summarize the outcome of a study aimed at comparing these two declarative approaches in solving combinatorial problems. The study has been originally presented in (Dovier, Formisano, & Pontelli 2005). We address a set of computationally hard problems—in particular, we mainly consider decision problems known to be NP-complete. We formalize each problem, both in CLP(FD) and in ASP, by taking advantage of the specific features available in each logical framework, and attempting to encode the various problems in the *most declarative* way possible. In particular, we adopt a *constraint-and-generate* strategy (Marriott & Stuckey 1998) for the construction of the CLP(FD) programs, while in ASP we exploit the natural *generate-and-test* approach (Baral 2003). Whenever possible, we make use of encodings of these problems that have been presented and widely accepted in the literature.

[1]This study has been originally presented in ICLP 2005 (Dovier, Formisano, & Pontelli 2005). This research is partially supported by MIUR projects PRIN05-015491 and FIRB03-RBNE03B8KK, and by NSF grants CNS-0220590, CNS-0454066, and HRD-0420407.

## Overview of the Computational Models

Constraint Logic Programming is a programming paradigm that is particularly well suited for encoding combinatorial optimization problems. CLP naturally merges two declarative paradigms: constraint satisfaction and logic programming. Let $\Sigma$ be a logic language signature $\Sigma = \langle \mathcal{F}, \mathcal{V}, \Pi \cup \Pi_C \rangle$, where $\mathcal{F}$ is a finite set of function and constant symbols, $\mathcal{V}$ is a denumerable collection of variables, and $\Pi \cup \Pi_C$ is a finite set of predicate symbols (with $\Pi$ and $\Pi_C$ disjoint). A *constraint* is a first-order formula over $\langle \mathcal{F}, \mathcal{V}, \Pi_C \rangle$. Typically, constraints are conjunctions of literals, e.g., $0 < X, X < 3, X + Y \neq 4$. Following the traditional logic programming notation, a comma indicates a conjunction, capital letters denote variables, and the symbol $:-$ denotes the implication $\leftarrow$. CLP lets a programmer use different classes of constraints and domains to encode problems. For combinatorial problems, it is common to use *finite domain constraints*, namely arithmetic constraints between arithmetic expressions, where each variable is associated to a finite domain of possible values. In this case the interpretation of variables, expressions, and constraints is over $\mathbb{Z}$.

A CLP program over $\Sigma$ is a finite set of rules of the form

$$p(s_1, \ldots, s_n) :- C, q_1(t^1_1, \ldots, t^1_{n_1}), \ldots, q_m(t^m_1, \ldots, t^m_{n_m})$$

where $C$ is a constraint, $s_i$ and $t^i_j$ are $(\mathcal{F}, \mathcal{V})$-terms, and $p, q_1, \ldots, q_m$ are predicate symbols of $\Pi$. Observe that a CLP program without constraints is a Prolog program.

CLP programs are commonly developed using a *constrain-and-generate* technique, where an initial deterministic phase imposes a number of constraints, and a non-deterministic phase generates/explores the solution space. In the `constraint` phase, in particular, a finite domain of values is assigned to each variable. For instance, the constraint `domain([A,B,C],1,5)` assigns the set of admissible values `{1,2,3,4,5}` to the variables `A`, `B`, and `C`. The built-in predicate `labeling` implements the non-deterministic *generate* phase through some form of search space exploration. Each time a variable is assigned a value, a deterministic propagation stage is executed, removing from the domains of the other variables those values that are incompatible with the assignments already performed. Various options, affecting, for instance, the variable selection criteria, the ordering of the attempted values, etc., can be used to guide the search. The main structure of a program using this programming style is the following:

```
solve_problem(X1,...,Xn) :-
        constraint(X1,...,Xn),
        labeling([options], X1,...,Xn).
```

A CLP(FD) system executes programs according to *goals* (i.e., conjunctions of literals) provided by the user. Given a program $P$ and a goal `?-solve_problem(X1,...,Xn)`, the system will determine the solutions in terms of instantiations of the variables `X1,...,Xn`.

Answer Set Programming is a logic programming paradigm that has its foundations in logic programming with negation as failure, under answer set semantics (Gelfond & Lifschitz 1988). ASP relies on a logic language signature $\Sigma = \langle \mathcal{F}, \mathcal{V}, \Pi \rangle$, where $\mathcal{F}$ is a finite set of constant symbols, $\mathcal{V}$ is a denumerable collection of variables, and $\Pi$ is a finite set of predicate symbols. The set of *terms* is $\mathcal{F} \cup \mathcal{V}$, and an atom is a formula of the type $p(t_1, \ldots, t_k)$, where $p \in \Pi$ and $t_1, \ldots, t_k$ are terms. An ASP program is a finite collection of rules, where rules are of the form

$$h :- a_1, \ldots, a_m, not\ b_1, \ldots, not\ b_n$$

where $h, a_1, \ldots, a_m, b_1, \ldots, b_n$ are atoms. Each program is viewed as a syntactic sugar representing the set of *ground instances* of its rules (computed using the constants in $\mathcal{F}$). A special type of rules are the so called *constraints*—rules with an empty head; in this case, the head of the rule is implicitly assumed to be *false*.

Given a ground program $P$, the answer set semantics characterizes the semantics of $P$ in terms of a collection of minimal models (called *answer sets*). A set of ground atoms $M$ is an answer set if $M$ is the unique minimal model of the program $P^M$, containing the rules $h :- a_1, \ldots, a_k$ such that $h :- a_1, \ldots, a_k, not\ b_1, \ldots, not\ b_h$ is in $P$ and $\{b_1, \ldots, b_h\} \cap M = \emptyset$. In ASP, each problem is modeled as a collection of rules, in such a way that the solutions to the problem correspond one-to-one to the answer sets of the program. An *ASP Solver* is a program that computes all the answer sets of a given ASP program. A solver can be seen as a theorem prover, or model builder, enhanced with several built-in heuristics to guide the exploration of the search space. Most ASP solvers rely on variations of the DPLL procedure (Davis, Logemann, & Loveland 1962). Such solvers are often equipped with a front-end that transforms a collection of non-propositional normal logic programming clauses (with limited use of function symbols) to a *finite* set of ground instances of such clauses.

## The Experimental Framework

The experimental studies, originally reported in (Dovier, Formisano, & Pontelli 2005), have been conducted using both CLP(FD) implementations and ASP solvers. The CLP(FD) programs have been designed for SICStus Prolog—and adapted to run also on GNU-Prolog, B-Prolog, and ECLiPSe. The ASP programs have been designed to be processed by *lparse*, the grounding preprocessor used by both the SMODELS and the CMODELS systems. The CMODELS system makes use of a SAT solver to compute answer sets—in our experiments we mainly used mChaff (Moskewicz *et al.* 2001) and Simo (Giunchiglia, Lierler, & Maratea 2006). SAT-based ASP solvers, such as CMODELS, take advantage of the *tightness* of the ASP programs (Lee & Lifschitz 2003). In presence of non-tight programs, CMODELS is forced to repeatedly call the SAT solver in order to reach a solution. This is done to discard those models of the program that are not answer sets, trying to avoid the introduction of a potentially exponential number of *loop formulae* (Lifschitz & Razborov 2006).

The study relied on a set of well-known computationally-hard problems: graph $k$-coloring, Hamiltonian circuit, Schur numbers, protein structure prediction in a 2D lattice, planning in a block world, generalized knapsack, and code design. Some of the proposed encodings have been drawn from the best proposals appeared in the literature, while others are novel solutions, developed for this project—e.g., the ASP implementation of the protein structure prediction problem and the blocks world planning in CLP(FD).

## Basic Encoding Schema

In most of the problems considered, we look for (the existence of) a function $f$, from a set of $N$ elements to a set of $K$ elements, fulfilling a collection of constraints that characterize the solutions of the problem. To fix the ideas, let us assume that the domain of $f$ is the set $\{1, \ldots, N\}$ while the range is the set $\{1, \ldots, K\}$.

In CLP(FD), problems of this kind are typically encoded by introducing a list `Vars` of $N$ variables whose domain is $\{1, \ldots, K\}$. Further constraints are imposed on `Vars` depending on the specific problem at hand. The resolution of such constraint satisfaction problem is then delegated to a *labeling* procedure, which explores the possible value-assignments for `Vars`.

The encoding style exploited in ASP is quite different. The function $f$ is represented by a (binary) predicate `pred_f`, defined in such a way that `pred_f(i,j)` holds if and only if $f(i) = j$. Thus, the general structure of this ASP-encoding is:

```
domain(1..n).                range(1..k).
1{pred_f(X,Y):range(Y)}1 :- domain(X).
```

Other rules must be added to properly define `pred_f`.

In our experiments we considered both the CPU usage time and memory consumption to compute the first solution, if any. The encoding of the various benchmarks and the complete results are reported in `www.dimi.uniud.it/dovier/CLPASP`.

## Summary of the Experimental Analysis

As first comparison between CLP(FD) and ASP, we investigated the programming effort required to encode a problem. From this perspective, we noticed that ASP provides more compact and more declarative encodings. In particular, the automatic grounding and the use of domain-restricted variables allow ASP programmers to avoid the explicit use of recursion in most situations. In general, in designing the CLP(FD) code, the programmer cannot easily ignore the specific inference strategies adopted by the CLP(FD) engine. The fact that CLP(FD) adopts a top-down, depth-first strategy affects the programmer's choices in encoding the algorithms. As a result, the amount of effort required to acquire good programming skills in the two paradigms is very different, in favor of ASP. A deeper understanding of specific "non-declarative" aspects of the CLP(FD) framework allows the programmer to develop more efficient, albeit more complex, solutions.

As far as performance is concerned, in a few cases the running times of SICStus and SMODELS are comparable, but in most of the cases CLP(FD) performs significantly better, especially whenever a solution exists (i.e., the "Yes" instances in Fig. 1). Nevertheless, there are cases (e.g., the code-generation problem) in which CLP(FD) fails in solving instances that are relatively easy for the ASP solvers. CMODELS is, in general, faster than SMODELS, and it competes better with CLP(FD) whenever the problem instance

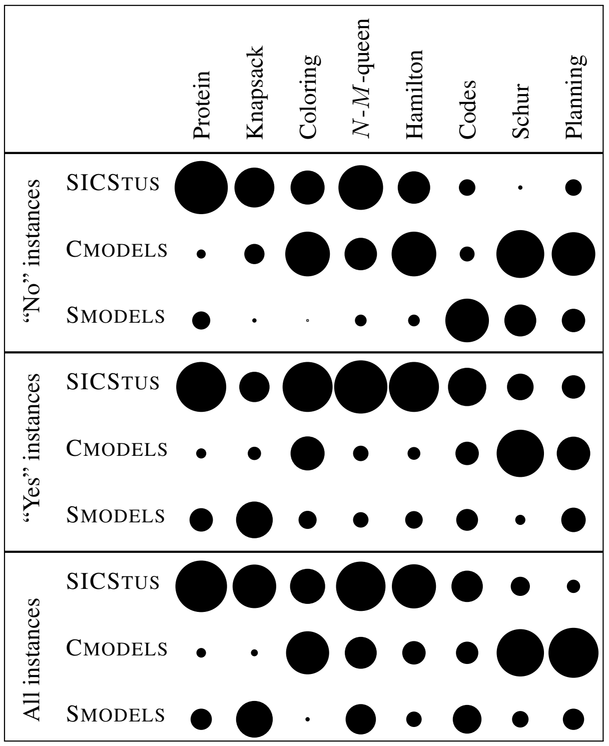

Figure 1: Relative performance of the solvers for each set of instances (the radii of the circles are proportional to the performance of the specific solver)

has no solutions. This can be justified by the fact that the programs we are using are mostly tight, and by the raw speed of the underlying SAT solver used by CMODELS.

We can draw some concrete indications that can be taken into account when choosing a paradigm to tackle a problem. We can summarize the main points as follows:

- The behavior of the solvers often depends on the existence of a solution. SICStus and SMODELS perform better on "Yes" instances, while CMODELS seems to be better suited for "No" instances;
- Graph-based problems have compact encodings in ASP, and the ASP solutions have acceptable and scalable performance, especially for the "No" instances (Fig. 1);
- Problems requiring more intense use of arithmetic and/or numbers (e.g., protein structure prediction and knapsack) are declaratively and efficiently handled by CLP(FD);
- For problems with no arithmetic, the exponential growth w.r.t. the input size is less of an issue for ASP (e.g., Schur numbers and planning).

Fig. 1 summarizes some of the results. For each problem instance, we compare the execution times of the three solvers; a score of 1 (0, $-1$) is assigned to the fastest (second fastest, slowest) solver. The scores of all instances of a problem are added, and this provides the radius of the circles in the Figure. Instances have been separated between "Yes" instances (they admit a solution) and "No" instances (no solutions).

As mentioned, our experiments suggest that the greater efficiency of CMODELS, w.r.t. SMODELS, originates from the speed of the underlying SAT solvers. Thus, one could wonder whether it is better to solve an instance of a hard combinatorial problem using a SAT-based ASP solver or directly employing a stand-alone SAT solver. The two alternatives require different approaches in modeling the problem, and different amounts of effort in encoding the problem. A comparison between SAT and ASP solvers is important, but goes beyond the current scope of our investigation. Nevertheless, we made a first step in this direction comparing the behavior of CMODELS and the behavior of the SAT solvers CMODELS uses, i.e., zChaff, Simo, and mChaff. The experiments we performed make use of instances of graph $k$-coloring and Hamiltonian circuit—these have been chosen as their encodings lead, respectively, to tight and non-tight programs. The instances to be processed by the SAT solvers have been generated by applying standard translations to SAT. The experiments show that the tightness of programs significantly affects the behavior of the solvers. Whenever a tight instance is tackled, it is often the case that SAT solvers outperform CMODELS. Non-tight programs are often efficiently solved by the ASP solver while all SAT solvers fail because of the growth in the size of the run-time image of the processes.

The better behavior of CMODELS in dealing with non-tight programs could be intuitively explained by observing that, while a SAT solver tries to solve the entire instance, CMODELS exploits a SAT solver to drive the construction of the answer sets through the discovery of loop formulae. This use of a SAT solver usually involves smaller SAT instances.

## Discussion

This work intends to develop a bridge between these two logic-based frameworks, in order to emphasize the strengths of each approach, and in favor of potential cross-fertilizations. This study also complements system benchmarking studies, recently proposed for both CLP(FD) (e.g., (Fernández & Hill 2000; Wallace *et al.* 2004)) and ASP solvers (e.g., (Anger, Schaub, & Truszczyński 2004)).

Bridges between the worlds have already been suggested, especially to target specific aspects of knowledge representation and reasoning. A system capable of integrating ASP reasoning within a Prolog shell has been presented in (Pontelli, Son, & Elkhatib 2006) and applied to support deep reasoning in question-answer systems. A reverse integration has also been explored, allowing an ASP solver to query a CLP(FD) system, to help supporting execution of aggregates in ASP (Elkabani, Pontelli, & Son 2004). Nevertheless, these are mostly ad-hoc integrations, where the line between the two systems has been manually drawn. Furthermore, the connection in these cases has been explored without regard to the issue of efficiency and adequacy.

## Conclusions

In this paper, we described an experimental study aimed at comparing the effectiveness of two logic-based declarative paradigms, i.e., CLP(FD) and ASP, on various classes of combinatorial problems. The aim of the study is to provide a better understanding of what makes one paradigm more suitable than the other in solving search and optimization problems. We provided what we feel are the most natural and declarative encodings of the various problems in the different paradigms, and we analyzed the performance of CLP(FD) and ASP solvers on these encodings.

In the future, we plan to extend our analysis to other problems and to other constraint (e.g., ILOG) and ASP solvers (e.g., ASSAT, DLV). In particular, we are interested in answering the following long term questions:

- Is it possible to formalize the domain and problem characteristics and employ them to choose the best paradigm?
- Is it possible to introduce strategies to split problem components and map them to cooperating solvers (using the best solver for each part of the problem)?

In particular, we are interested in identifying those contexts where the ASP solvers perform significantly better than CLP(FD). It seems reasonable to expect this behavior, for instance, whenever incomplete information comes into play (e.g., modeling conformant planning).

## References

Anger, C.; Schaub, T.; and Truszczyński, M. 2004. ASPARAGUS - the Dagstuhl Initiative. *ALP Newsletter* 17(3).

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Comm. of the ACM* 5:394–397.

Dovier, A.; Formisano, A.; and Pontelli, E. 2005. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. *Int. Conference on Logic Progr.*, 67–82. Springer.

Elkabani, I.; Pontelli, E.; and Son, T. C. 2004. Smodels with CLP and its applications. *Int. Conf. Logic Progr.*, Springer, 73–89.

Fernández, A., and Hill, P. M. 2000. A comparative study of eight constraint programming languages over the Boolean and Finite Domains. *Constraints* 5:275–301.

Gelfond, M., and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. *Int. Conf. on Logic Progr.*, MIT Press, 1070–1080.

Giunchiglia, E. et al. 2006. Answer set programming based on propositional satisfiability. *J. Automat. Reason.* 36(4):345–377.

Jaffar, J., and Maher, M. J. 1994. Constraint Logic Programming: A Survey. *J. of Logic Progr.* 19/20:503–581.

Lee, J., and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs. *Int. Conf. Logic Progr.*, Springer, 451–465.

Lifschitz, V., and Razborov, A. A. 2006. Why are there so many loop formulas? *ACM-TOCL* 7(2):261–268.

Marek, V., and Truszczynski, M. 1991. Autoepistemic Logic. *JACM* 38(3):588–619.

Marriott, K., and Stuckey, P. J. 1998. *Programming with Constraints*. The MIT Press.

Moskewicz, M. W. et al. 2001. Chaff: Engineering an efficient SAT solver. *DAC-01*, ACM Press, 530–535.

Pontelli, E.; Son, T.; and Elkhatib, O. 2006. A Tool for Knowledge Base Integration and Querying. *AAAI Spring Symposium*.

Simons, P. 2000. *Extending and Implementing the Stable Model Semantics*. Ph.D. Dissertation, Helsinki Univ. Technology.

Wallace, M. et al. 2004. On benchmarking constraint logic programming platforms. *Constraints* 9:5–34.