aspartame: Solving Constraint Satisfaction Problems with Answer Set Programming*

M. Banbara[3], M. Gebser[1,6], K. Inoue[4], M. Ostrowski[6], A. Peano[5], T. Schaub[6,2**], T. Soh[3], N. Tamura[3], and M. Weise[6]

[1]Aalto University Helsinki [2]INRIA Rennes [3]Kobe University [4]NII Tokyo
[5]University of Ferrara [6]University of Potsdam

Abstract. Encoding finite linear CSPs as Boolean formulas and solving them by using modern SAT solvers has proven to be highly effective by the award-winning *sugar* system. We here develop an alternative approach based on ASP that serves two purposes. First, it provides a library for solving CSPs as part of an encompassing logic program. Second, it furnishes an ASP-based CP solver similar to *sugar*. Both tasks are addressed by using first-order ASP encodings that provide us with a high degree of flexibility, either for integration within ASP or for easy experimentation with different implementations. When used as a CP solver, the resulting system *aspartame* re-uses parts of *sugar* for parsing and normalizing CSPs. The obtained set of facts is then combined with an ASP encoding that can be grounded and solved by off-the-shelf ASP systems. We establish the competitiveness of our approach by empirically contrasting *aspartame* and *sugar*.

1 Introduction

Encoding finite linear Constraint Satisfaction Problems (CSPs; [2]) as propositional formulas and solving them by using modern solvers for Satisfiability Testing (SAT; [3]) has proven to be a highly effective approach by the award-winning *sugar*[1] system. The CP solver *sugar* reads a CSP instance and transforms it into a propositional formula in Conjunctive Normal Form (CNF). The translation relies on the order encoding [4, 5], and the resulting CNF formula can be solved by an off-the-shelf SAT solver.

In what follows, we elaborate upon an alternative approach based on Answer Set Programming (ASP; [6]) and present the resulting *aspartame*[2] framework, serving two purposes. First, *aspartame* provides a library for solving CSPs as part of an encompassing logic program. Second, it constitutes an ASP-based CP solver similar to *sugar*. The major difference between *sugar* and *aspartame* rests upon the implementation of the translation of CSPs into Boolean constraint problems. While *sugar* implements a translation into CNF in Java, *aspartame* starts with a translation into a set of facts.[3] When

* This paper is a greatly revised version of the workshop paper [1]. The work was funded by [1]AoF (251170), [6]DFG (SCHA 550/10-1), [5]5x1000 (UNIFE 2011), and [3]JSPS (KAKENHI 15K00099).

** Affiliated with Simon Fraser University, Canada, and IIIS Griffith University, Australia.

[1] `http://bach.istc.kobe-u.ac.jp/sugar`

[2] `http://www.cs.uni-potsdam.de/aspartame`

[3] When used as CP solver, *aspartame* re-uses *sugar*'s front-end for parsing and normalizing (non-linear) CSPs. Also, we extended *sugar* to produce a fact-based representation.

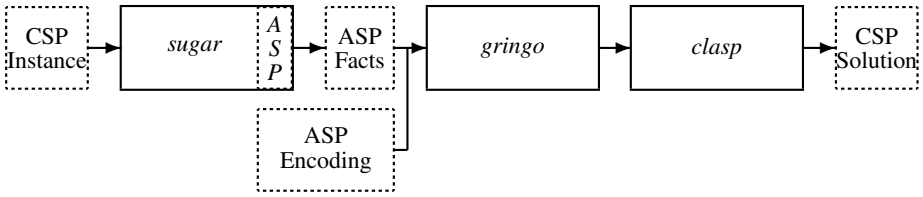

Fig. 1. Architecture of *aspartame*.

used as a library, this set of facts (representing the CSP) must be supplied by the user.[4] In turn, these facts are combined with a general-purpose ASP encoding for CP solving (also based on the order encoding), which is subsequently instantiated by an off-the-shelf ASP grounder, in our case *gringo*. The resulting propositional logic program is then solved by an off-the-shelf ASP solver (here *clasp*). The architecture of *aspartame* is given in Figure 1.

The high-level approach of ASP has obvious advantages. First, instantiation is done by general-purpose ASP grounders rather than dedicated implementations. Second, the elaboration tolerance of ASP allows for easy maintenance and modifications of encodings. And finally, it is easy to experiment with novel or heterogeneous encodings. However, the question is whether the high-level approach of *aspartame* matches the performance of the more dedicated *sugar* system. We empirically address this question by contrasting the performance of both CP solvers, while fixing the back-end solver to *clasp*, used as both a SAT and an ASP solver.

From an ASP perspective, we gain insights into advanced modeling techniques for solving CSPs. The ASP encoding implementing CP solving with *aspartame* has the following features:

- usage of function terms to abbreviate structural subsums
- avoidance of (artificial) intermediate Integer variables (to split sum expressions)
- a collection of encodings for the *alldifferent* constraint

In the sequel, we assume some familiarity with ASP, its semantics as well as its basic language constructs. A comprehensive treatment of ASP can be found in [6], one oriented towards ASP solving is given in [7]. Our encodings are given in the language of *gringo* 4. Although we provide essential definitions of CSPs in the next section, we refer the reader to the literature [2] for a broader perspective.

2 Background

A *Constraint Satisfaction Problem* (CSP) is given by a pair $(\mathcal{V}, \mathcal{C})$ consisting of a set $\mathcal{V}$ of *variables* and a set $\mathcal{C}$ of *constraint clauses*. Every variable $x \in \mathcal{V}$ has an associated finite *domain* $D(x)$ such that either $D(x) = \{\top, \bot\}$ or $\emptyset \subset D(x) \subseteq \mathbb{Z}$; x is a *Boolean variable* if $D(x) = \{\top, \bot\}$, and an *Integer variable* otherwise. We denote the set of Boolean and Integer variables in $\mathcal{V}$ by $\mathcal{B}(\mathcal{V})$ and $\mathcal{I}(\mathcal{V})$, respectively. A constraint clause $C \in \mathcal{C}$ is a set of literals, and a *literal* is of the form e or $\overline{e}$, where e is either a Boolean

[4] This will be integrated into *gringo*'s input language in the near future.

variable in $\mathcal{B}(\mathcal{V})$, a linear inequality, or an *alldifferent* constraint. A *linear inequality* is an expression $\sum_{1 \leq i \leq n} a_i x_i \leq m$ in which m as well as all a_i for $1 \leq i \leq n$ are Integer constants and $x_1, \ldots, x_n$ are Integer variables in $\mathcal{I}(\mathcal{V})$. An *alldifferent* constraint (cf. [8]) applies if a subset $\{x_1, \ldots, x_n\}$ of Integer variables in $\mathcal{I}(\mathcal{V})$ is assigned to distinct values in their respective domains.[5]

Given a CSP $(\mathcal{V}, \mathcal{C})$, a *variable assignment* v is a (total) mapping $v : \mathcal{V} \to \bigcup_{x \in \mathcal{V}} D(x)$ such that $v(x) \in D(x)$ for every $x \in \mathcal{V}$. A Boolean variable $x \in \mathcal{B}(\mathcal{V})$ is *satisfied* wrt v if $v(x) = \top$. Likewise, a linear inequality $\sum_{1 \leq i \leq n} a_i x_i \leq m$ is *satisfied* wrt v if $\sum_{1 \leq i \leq n} a_i v(x_i) \leq m$ holds. An *alldifferent* constraint over subsets $\{x_1, \ldots, x_n\}$ of $\mathcal{I}(\mathcal{V})$ is *satisfied* wrt v if $v(x_i) \neq v(x_j)$ for all $1 \leq i < j \leq n$, respectively. Any Boolean variable, linear inequality, or *alldifferent* constraint that is not satisfied wrt v is *unsatisfied* wrt v. A constraint clause $C \in \mathcal{C}$ is *satisfied* wrt v if there is some literal $e \in C$ (or $\overline{e} \in C$) such that e is satisfied (or unsatisfied) wrt v. The assignment v is a *solution* for $(\mathcal{V}, \mathcal{C})$ if every $C \in \mathcal{C}$ is satisfied wrt v.

For illustration, consider a CSP $(\mathcal{V}, \mathcal{C})$ with Boolean and Integer variables $\mathcal{B}(\mathcal{V}) = \{b\}$ and $\mathcal{I}(\mathcal{V}) = \{x, y, z\}$, where $D(x) = D(y) = D(z) = \{1, 2, 3\}$, and constraint clauses $\mathcal{C} = \{C_1, C_2, C_3\}$ as follows:

$$C_1 = \{\mathit{alldifferent}(x, y, z)\} \tag{1}$$

$$C_2 = \{b, 4x - 3y + z \leq 0\} \tag{2}$$

$$C_3 = \{\overline{b}, -4x + 3y \leq -6\} \tag{3}$$

The *alldifferent* constraint in C_1 requires values assigned to x, y, and z to be mutually distinct. Assignments v satisfying the linear inequality $4x - 3y + z \leq 0$ in C_2 include $v(x) = 2$, $v(y) = 3$, and $v(z) = 1$ or $v(x) = 1$, $v(y) = 3$, and $v(z) = 2$, while assignments for the constraint $-4x + 3y \leq -6$ include $v(x) = 3$, $v(y) = 2$, and $v(z) = 1$ or $v(x) = 3$, $v(y) = 1$, and $v(z) = 2$. In view of the Boolean variable b, whose value allows for "switching" between the constraints in C_2 and C_3, we obtain the four solutions $v_1, \ldots, v_4$ for $(\mathcal{V}, \mathcal{C})$ shown alongside.

	b	x	y	z
v_1	$\bot$	2	3	1
v_2	$\bot$	1	3	2
v_3	$\top$	3	2	1
v_4	$\top$	3	1	2

3 The *aspartame* Approach

For using *aspartame* as a CP solver, we extended the front-end of the *sugar* system by an output component representing CSPs in terms of ASP facts. The latter also constitute the CSP instances when using *aspartame* as library. As usual, the resulting facts can then be combined with a first-order encoding processable with off-the-shelf ASP systems. In what follows, we describe *aspartame*'s fact format and we present dedicated ASP encodings utilizing function terms to capture substructures in CSP instances.

Fact Format. Facts express the variables and constraints of a CSP instance in the syntax of ASP grounders like *gringo*. Their format is easiest explained on the CSP from

[5] Linear and non-linear inequalities relying on further comparison operators, such as $<$, $>$, $\geq$, $=$, and $\neq$, can be converted into the considered format via appropriate replacements [5]. Moreover, note that we here limit the constraints to the ones that are directly, i.e., without normalization by *sugar*, supported in our prototypical ASP encodings shipped with *aspartame*.

```
1  var(bool,b).      var(int,(x;y;z),range(1,3)).

3  constraint(1,global(alldifferent,arg(x,arg(y,arg(z,nil))))).
4  constraint(2,b).
5  constraint(2,op(le,op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),0)).
6  constraint(3,op(neg,b)).
7  constraint(3,op(le,op(add,op(mul,-4,x),op(mul,3,y)),-6)).
```

Listing 1. Facts representing the CSP from Section 2.

Section 2, whose fact representation is shown in Listing 1. While facts of the predicate `var`/2 provide labels of Boolean variables, like b, the predicate `var`/3 includes a third argument for declaring the domains of Integer variables, like x, y, and z. Domain declarations rely on function terms `range(`l`,`u`)`, standing for Integer intervals $[l, u]$. While one term, `range(1,3)`, suffices for the common domain $\{1, 2, 3\}$ of x, y, and z, in general, several intervals can be specified (via separate facts) to form non-continuous domains. Note that the interval format for Integer domains offers a compact fact representation of domains; e.g., the single term `range(1,10000)` captures a domain of 10000 elements. Furthermore, the usage of meaningful function terms avoids any need for artificial labels to refer to domains or parts thereof.

The literals of constraint clauses are also represented by means of function terms. In fact, the second argument of `constraint`/2 in Line 3 of Listing 1 stands for $\mathit{alldifferent}(x, y, z)$ from the constraint clause C_1 in (1), identified by the first argument of `constraint`/2. Since each fact of predicate `constraint`/2 is supposed to describe a single literal only, constraint clause identifiers establish the connection between individual literals of a clause. The more complex term of the form `op(le,`Σ`,`m`)` in Line 5 stands for a linear inequality $\Sigma \leq m$. In particular, the inequality $4x-3y+z \leq 0$ from C_2 is represented by nested `op(add,`Σ`,`ax`)` terms whose last argument ax and deepest Σ part are of the form `op(mul,`a`,`x`)`; such nesting corresponds to the precedence $(((4*x)+(-3*y))+(1*z)) \leq 0$. The representation by function terms captures linear inequalities of arbitrary arity and, as with Integer intervals, associates (sub)sums with canonical labels.

The function term expressing the *alldifferent* constraint includes an argument list of the form `arg(`x_1`,arg(...,arg(`x_n`,nil)...))`, in which $x_1, \ldots, x_n$ refer to Integer variables. In Line 3 of Listing 1, an *alldifferent* constraint over arguments $\boldsymbol{x}$ is declared via `global(alldifferent,`$\boldsymbol{x}$`)`; at present, `alldifferent` is a fixed keyword in facts used by *aspartame*, but support for other kinds of global constraints can be added in the future.

First-Order Encoding. In addition to a dedicated output component of *sugar* for generating ASP facts, *aspartame* comes with several alternative first-order ASP encodings for solving CSP instances. In the following, we first describe a basic encoding that implements the order encoding techniques [5, 9] concisely, and then present optimizations and extensions for the *alldifferent* constraints. We also show that these *aspartame* encodings can be used as library for solving CSPs and Constraint Optimization Problems. Due to lack of space, *aspartame* encodings presented here are restricted to unary constraint clauses and stripped off capacities for handling Boolean variables.

```
1  var(V) :- var(int,V,_).
2  lb(V,@getLB(V)) :- var(V).
3  ub(V,@getUB(V)) :- var(V).

5  global(I,global(Func,Args)) :- constraint(I,global(Func,Args)).
6  wsum(@sortWsum(L))          :- constraint(I,L), not global(_,L).
7  alldiff(Args)               :- global(_,global(alldifferent,Args)).

9  exp(E)     :- wsum(op(_,E,_)).
10 exp(E1;E2) :- exp(op(add,E1,E2)).
11 unary_exp(op(mul,A,V)) :- exp(op(mul,A,V)).

13 inf(op(mul,A,V),A*LB)  :- exp(op(mul,A,V)), A > 0, lb(V,LB).
14 inf(op(mul,A,V),A*UB)  :- exp(op(mul,A,V)), A < 0, ub(V,UB).
15 inf(op(add,E1,E2),A+B) :- exp(op(add,E1,E2)), inf(E1,A), inf(E2,B).

17 sup(op(mul,A,V),A*UB)  :- exp(op(mul,A,V)), A > 0, ub(V,UB).
18 sup(op(mul,A,V),A*LB)  :- exp(op(mul,A,V)), A < 0, lb(V,LB).
19 sup(op(add,E1,E2),A+B) :- exp(op(add,E1,E2)), sup(E1,A), sup(E2,B).
```

Listing 2. Auxiliary predicates

```
1  { p(V,A) : A = @getDom(V) } :- var(V).
2  :- p(V,A), not p(V,B), B = @getSimpGT(V,A), A < UB, ub(V,UB).
3  :- not p(V,UB), var(V), ub(V,UB).
```

Listing 3. Encoding of Integer variables

Basic Encoding. Listing 2 shows common auxiliary predicates shared by *aspartame* encodings. Given a CSP instance, for each Integer variable `V`, the domain values of the variables are kept in a *lua* table, and then the lower and upper bounds of each `V` are calculated via *lua* and captured in the second arguments of `lb`/2 and `ub`/2 respectively in Line 1–3. Each literal of a constraint clause is classified into a *alldifferent* constraint expressed by `alldiff`/1 or a linear inequality by `wsum`/1 in Line 5–7. The identifier in the first argument of `constraint`/2 is removed in Line 6–7, since the *aspartame* encoding presented here is restricted to unary constraint clauses. For each linear inequality $\sum_{i=1}^{n} a_i x_i \leq c$, in Line 6, Σ is sorted in descending order of $|D(x_i)|$ via *lua*, and then $\Sigma \leq c$ is captured in the argument of predicate `wsum`/1. Each sum Σ in `op(_,`Σ`,_)` is decomposed into (sub)sums in Line 9–10, and then the lower and upper bounds of them are calculated and captured in the second arguments of `inf`/2 and `sup`/2 respectively in Line 13–19. In Line 11, a predicate `unary_exp(`e`)` is generated if e is an unary expression of the form `op(mul,`a_i`,`x_i`)`.

Listing 3 gives our encoding of Integer variables. For each variable `V` and each domain value `A` in $D(\texttt{V})$, we introduce a predicate `p(V,A)` expressing that `V` $\leq$ `A`. The truth-assignments of the instances of `p`/2 are encoded by the choice rule in Line 1. The constraints in Line 2 and 3 ensure that each variable `V` has exactly one value in $D(\texttt{V})$. Note that the *lua* function `getDom(V)` returns $D(\texttt{V})$, and `getSimpGT(`x`,`c`)` $= \min\{d \in D(x) \mid d > c\}$. For illustration, consider an Integer variable $x \in \{2,3,4,5,6\}$ represented by `var(int,x,range(2,6))` as an ASP fact. The resulting propositional logic program and its answer sets obtained by *clasp* are as follows.

```
1  wsum(op(le,X,K-B*D)) :-
2          not unary(X) : opt_binary == 1;
3          wsum(op(le,op(add,X,op(mul,B,Y)),K)),
4          not p(Y,C) : B > 0, C = @getSimpLE(Y,D-1);
5              p(Y,D) : B < 0;
6          D = @getDomOpt(Y,B,K-Sup,K-Inf), inf(X,Inf), sup(X,Sup).

8  wsum(op(le,op(mul,B,Y),K-Inf)) :-
9          wsum(op(le,op(add,X,op(mul,B,Y)),K)),
10         K-Inf < Sup, inf(X,Inf), sup(op(mul,B,Y),Sup).
```

Listing 4. Encoding of $\sum_{i=1}^{n} a_i x_i \leq c$ $(n \geq 2)$

```
{p(x,2),p(x,3),p(x,4),p(x,5),p(x,6)}.
:- p(x,2),not p(x,3).
:- p(x,3),not p(x,4).
:- p(x,4),not p(x,5).
:- p(x,5),not p(x,6).
:- not p(x,6).
```

answer sets	interpretation
{p(x,6)}	$x = 6$
{p(x,5),p(x,6)}	$x = 5$
{p(x,4),p(x,5),p(x,6)}	$x = 4$
{p(x,3),p(x,4),p(x,5),p(x,6)}	$x = 3$
{p(x,2),p(x,3),p(x,4),p(x,5),p(x,6)}	$x = 2$

Constraints are encoded into clauses expressing conflict regions instead of conflict points. Especially, for any linear inequality $\sum_{i=1}^{n} a_i x_i \leq c$, the following holds [9].

$$\sum_{i=1}^{n} a_i x_i \leq c \Longleftrightarrow \begin{cases} (x_1 \leq \lfloor c/a_1 \rfloor) & (n = 1, a_1 > 0) \quad (4) \\ \neg(x_1 \leq \lceil c/a_1 \rceil - 1) & (n = 1, a_1 < 0) \quad (5) \\ \bigwedge_{d \in D(x_n)} \Big((x_n \leq d - 1) \vee \sum_{i=1}^{n-1} a_i x_i \leq c - a_n d \Big) & (n \geq 2, a_n > 0) \quad (6) \\ \bigwedge_{d \in D(x_n)} \Big(\neg(x_n \leq d) \vee \sum_{i=1}^{n-1} a_i x_i \leq c - a_n d \Big) & (n \geq 2, a_n < 0) \quad (7) \end{cases}$$

From this mathematical representation, we can define a recursive encoding procedure by considering $\sum_{i=1}^{n-1} a_i x_i \leq c - a_n d$ as the recursive call and also replacing the comparison of the form $x \leq c$ with the translation $\|x \leq c\|$. The translation $\|x \leq c\|$ is defined as $\top$ if $c \geq \max(D(x))$, $\bot$ else if $c < \min(D(x))$, and otherwise $\mathtt{p}(x, d)$ where $d = \max\{d' \in D(x) \mid d' \leq c\}$.

This encoding procedure can be optimized by considering the validity and inconsistency of the recursive part $\sum_{i=1}^{n-1} a_i x_i \leq c - a_n d$, which can reduce the number of iterations and clauses. The validity and inconsistency of the recursive part can be captured by inequalities $c - a_n d \geq \sup(\sum_{i=1}^{n-1} a_i x_i)$ and $c - a_n d < \inf(\sum_{i=1}^{n-1} a_i x_i)$ respectively, where $\inf(\Sigma)$ and $\sup(\Sigma)$ indicate the lower and upper bounds of linear expression Σ respectively. When the recursive part is valid, the clause containing it is unnecessary. When it is inconsistent, the literal of the recursive part can be removed, and moreover only one such clause is sufficient. Based on the observations above, we present a recursive ASP encoding of linear inequalities.

Encoding of $\sum_{i=1}^{n} a_i x_i \leq c$ $(n \geq 2)$ corresponding to (6) and (7) is presented in Listing 4. For a given linear inequality `op(le,op(add,X,op(mul,B,Y)),K)`, in the first rule, a recursive part `wsum(op(le,X,K-B*D))` is generated for every domain value `D` in $D(\mathtt{Y})$ calculated via *lua* such that the recursive part becomes neither valid nor inconsistent, if "`not p(Y,C)`" holds when `B` $>$ 0 (or `p(Y,D)` holds when `B` $<$ 0). Intuitively, "`not p(Y,C)`" expresses `Y` $\geq$ `D` because of

```
1 :- wsum(op(le,op(mul,A,X),K)), Inf <= K, K < Sup,
2           inf(op(mul,A,X),Inf), sup(op(mul,A,X),Sup),
3           not p(X,B) : A > 0; p(X,B) : A < 0;
4           B = @getLE(X,A,K).

6 :- wsum(op(le,op(mul,A,X),K)), Inf > K, inf(op(mul,A,X),Inf).
```

Listing 5. Encoding of $a_1x_1 \leq c$

```
1 :- wsum(op(le,op(add,op(mul,A,X),op(mul,B,Y)),K)),
2         not p(Y,C) : B > 0 , C = @getSimpLE(Y,D-1);
3             p(Y,D) : B < 0;
4         not p(X,E) : A > 0;
5             p(X,E) : A < 0;
6         D = @getDomOpt(Y,B,K-Sup,K-Inf), inf(op(mul,A,X),Inf), sup(op(mul,A,X),Sup),
7         E = @getLE(X,A,K-B*D).
```

Listing 6. Encoding of $\sum_{i=1}^{n} a_i x_i \leq c$ $(n = 2)$

`getSimpLE(`x`,`c`)` $= \max\{d \in D(x) \mid d \leq c\}$. Note that the conditional literal in Line 2 is ignored, since the constant `opt_binary` is set to 0 initially. In the second rule, only one `wsum(op(le,op(mul,B,Y),K-Inf))` corresponding to $a_n x_n \leq c - \inf(\sum_{i=1}^{n-1} a_i x_i)$ is generated if there exists at least one domain value in $D($`Y`$)$ such that the recursive part becomes inconsistent.

Encoding of $a_1x_1 \leq c$ corresponding to (4) and (5) is presented in Listing 5. For a given linear inequality `op(le,op(mul,A,X),K)`, if it is neither valid nor inconsistent, the first rule calculates a bound `B` in $D($`X`$)$ via *lua* and ensures that `p(X,B)` holds when `A` > 0 (or `p(X,B)` does not hold when `A` < 0). The *lua* function `getLE(`x`,`a`,`c`)` is defined as `getSimpLE(`x`,`$\lfloor c/a \rfloor$`)` if $a > 0$, otherwise `getSimpLE(`x`,`$\lceil c/a \rceil - 1$`)`. If inconsistent, the second rule ensures that `wsum(op(le,op(mul,A,X),K))` never holds.

We refer to the encoding of Listing 2–5 as *basic encoding*. This encoding can concisely implement CP solving based on the order encoding techniques by utilizing the feature of function terms. Moreover, it proposes an alternative approach to splitting sum expressions. In fact, the basic encoding splits them by generating the instances of predicate `wsum`/1 during recursive encoding, rather than by introducing intermediate Integer variables during preprocessing like a *sugar*'s CSP-to-CSP translation. It is noted that global constraints such as *alldifferent* and *cumulative* are first translated into linear inequalities by *sugar*'s front-end and then encoded by the basic encoding.

Optimizations and Extensions. The basic encoding generates some redundant clauses for linear inequalities of size two. Consider $x + y \leq 7$ represented by a function term `op(le,op(add,op(mul,1,x),op(mul,1,y)),7)`, where $D(x) = D(y) = \{2, 3, 4, 5, 6\}$. The resulting propositional logic program is as follows.

```
:- not p(y,5). :- not p(x,5).
wsum(op(le,op(mul,1,x),2)) :- not p(y,4). :- wsum(op(le,op(mul,1,x),2)),not p(x,2).
wsum(op(le,op(mul,1,x),3)) :- not p(y,3). :- wsum(op(le,op(mul,1,x),3)),not p(x,3).
wsum(op(le,op(mul,1,x),4)) :- not p(y,2). :- wsum(op(le,op(mul,1,x),4)),not p(x,4).
```

The intermediate instances of `wsum`/1 are redundant and can be removed. This issue can be fixed by the *optimized encoding* which is an extension of the basic encoding by adding only the one rule of Listing 6 and by setting the constant `opt_binary` to 1. The rule of Listing 6 represents the special case of the first rule in Listing 4 for

```
1  alldiff :- alldiff(_).
2  val(V,A) :- var(int,V,_), p(V,A), not p(V,@getSimpLE(V,A-1)), alldiff.

4  alldiffArg(arg(F,A),1,F,A) :- alldiff(arg(arg(F,A),nil)).
5  alldiffArg(N,I+1,F,A) :- alldiffArg(N,I,_,arg(F,A)).

7  alldiffRange(A,LB,UB) :- alldiff(arg(A,nil)),
8          LB = #min {L,V : var(int,V,range(L,_)), alldiffArg(A,_,V,_)},
9          UB = #max {U,V : var(int,V,range(_,U)), alldiffArg(A,_,V,_)}.
```

Listing 7. Auxiliary predicates for the *alldifferent* constraints

```
1  :- alldiffRange(CI,LB,UB), X = LB..UB, val(V1,X), val(V2,X),
2     alldiffArg(CI,_,V1,_), alldiffArg(CI,_,V2,_), V1 < V2.
```

Listing 8. alldiffA encoding

$\sum_{i=1}^{n} a_i x_i \leq c$ $(n = 2)$ and does not generate any intermediate instances of `wsum`/1. However, we keep generating such intermediate instances for $n > 2$ because they can be shared by different linear inequalities and can be effective in reducing the number of clauses. For the above example, the optimized encoding generates the following.

```
:- not p(y,5). :- not p(x,5).
:- not p(y,4), not p(x,2). :- not p(y,3), not p(x,3). :- not p(y,2), not p(x,4).
```

To extend our approach, we present four different encodings for the *alldifferent* constraints: *alldiffA*, *alldiffB*, *alldiffC*, and *alldiffD*. Listing 7 shows common auxiliary predicates shared by these encodings. For each variable `V` and domain value `A` in $D(\mathtt{V})$, we introduce a predicate `val(V,A)` that expresses $\mathtt{V} = \mathtt{A}$ in Line 2. For each *alldifferent* constraint, its argument list is decomposed into a flat representation of the variables by `alldiffArg`/4 in Line 4–5, and then the lower and upper bounds of the variables are calculated and captured in the second and third arguments of `alldiffRange`/3 respectively. Note that, except in *alldiffD*, the *alldifferent* constraints are encoded by using the predicate `val`/2 [10]. The alldiffA encoding is the simplest one presented in listing 8. It consists of only one integrity constraint forbidding that two distinct variables have exactly the same domain value. The alldiffB encoding in listing 9 does exactly the same as alldiffA, but uses a cardinality constraint. The alldiffC encoding in listing 10 is more mature. We use a fixed ordering of the identifiers of the variables. In Line 1–2, whenever a variable `V` has been assigned to a value `X`, we derive a `seen` predicate for every variable whose identifier is less than that of `V` for the value `X`. In Line 3 we forbid that a variable `V` has the same value as a variable whose identifier is greater than that of V. The alldiffD encoding, not presented due to lack of space, uses hall intervals [11]. Therefore, for each variable `V` and each interval $[\mathtt{A}, \mathtt{B}]$, we introduce a Boolean variable `p(V,A,B)` that is true iff `V` is in the interval $[\mathtt{A}, \mathtt{B}]$. Moreover, a linear inequality is added to constrain the number of variables having a value in this interval to $\mathtt{B} - \mathtt{A}$. We parametrized the alldiffD encoding with a maximal interval size.

Using *aspartame* encodings as a library. Let us consider the Two Dimensional Strip Packing (2sp) problems. Given a set of n rectangles and one large rectangle (called strip), the goal of the 2sp problem is to find the minimum strip height such that all rectangles are packed into the strip without overlapping.

A 2sp instance is given as a set of facts consisting of `width(`W`)` and `r(`i`,`w_i`,`h_i`)` for $1 \leq i \leq n$. The fact `width(`W`)` expresses that the width of the strip is W, and

```
1 :- alldiffRange(CI,LB,UB),
2    X = LB..UB, 2{val(V,X) : alldiffArg(CI,_,V,_)}.
```

Listing 9. alldiffB encoding

```
1 seen(CI,I-1,X) :- alldiffArg(CI,I,V,_), 1 < I, val(V,X).
2 seen(CI,I-1,X) :- seen(CI,I,X), 1 < I.
3 :- alldiffArg(CI,I,V,_), val(V,X), seen(CI,I,X).
```

Listing 10. alldiffC encoding

`r(`i`,`w_i`,`h_i`)` expresses the rectangle i with a width of w_i and a height of h_i. Using *aspartame* we introduce a pair of Integer variables (x_i, y_i) that express the position of lower left coordinates of each rectangle i, and then enforcing non-overlapping constraints $(x_i + w_i \leq x_j) \vee (x_j + w_j \leq x_i) \vee (y_i + h_i \leq y_j) \vee (y_j + h_j \leq y_i)$ for every two different rectangles i and j ($i < j$). For example, $x_i + w_i \leq x_j$ ensures that the rectangles i is located to the lefthand-side of j.

An ASP encoding for solving 2sp problems is shown in Listing 11. The predicates except `le`/3, `width`/1, and `r`/3 are defined in *aspartame* encodings. The Integer variable `height` with an initial range from `lb` to `ub`, in Line 3–4, is an objective variable that we want to minimize. The predicate `le(`x`,`c`,`y`)` is intended to express $x + c \leq y$ where x and y are Integer variables and c is an Integer constant. The non-overlapping constraints can be concisely expressed by using cardinality constraints. In preliminary experiments (not presented in this paper), we confirmed that this encoding can be highly competitive in performance to a SAT-based approach for solving 2sp problems [12].

4 The *aspartame* System

As mentioned, *aspartame* re-uses *sugar*'s front-end for parsing and normalizing CSPs. Hence, it accepts the same input formats, viz. XCSP[6] and *sugar*'s native CSP format[7]. For this, we implemented an output hook for *sugar* that provides us with the resulting CSP instance in *aspartame*'s fact format. This format can also be used for directly representing linear arithmetic constraints within standard ASP encodings used for Constraint ASP (CASP; [13–16]). In both cases, the resulting facts are then used for grounding a dedicated ASP encoding (via the ASP grounder *gringo*). In turn, the resulting propositional logic program is passed to the ASP solver *clasp* that returns an assignment, representing a solution to the original CSP instance.

Our empirical analysis considers all instances of `GLOBAL` categories in the 2009 CSP Competition[6]. We ran them on a cluster of Linux machines equipped with dual Xeon E5520 quad-core 2.26 GHz processors and 48 GB RAM. We separated grounding and solving times, and imposed on each a limit of 1800s and 16GB. While we count a grounding timeout as 1800s, we penalize unsuccessful solving with 1800s if either solving or grounding does not finish in time

At first, we analyze the difference between the *basic encoding* and its refinements from the previous section. To this end, Table 1 contrasts the results obtained from dif-

[6] `http://www.cril.univ-artois.fr/CPAI09`

[7] `http://bach.istc.kobe-u.ac.jp/sugar/package/current/docs/syntax.html`

```
1  var(int,   x(I), range(0,W-X))  :- r(I,X,Y), width(W).
2  var(int,   y(I), range(0,ub-Y)) :- r(I,X,Y).
3  var(int, height, range(lb,ub)).
4  objective(minimize, height).

6  1 { le(x(I),XI,x(J)) ; le(x(J),XJ,x(I)) ; le(y(I),YI,y(J)) ; le(y(J),YJ,y(I)) } :-
7          r(I,XI,YI), r(J,XJ,YJ), I < J.
8  le(y(I),Y,height) :- r(I,X,Y).

10 wsum(op(le,op(add,op(mul,1,X),op(mul,-1,Y)),-C)) :- le(X,C,Y).
```

Listing 11. Encoding of 2sp problems

	basic		*optimized*		*optimizednosplit*		*sugar*	
Benchmark Class	trans	solve(to^t,to)	trans	solve(to^t,to)	trans	solve(to^t,to)	trans	solve(to^t,to)
CabinetStart1(40)	1800	1800 (40, 40)	53	20 (0, 0)	8	2 (0, 0)	4	12 (0, 0)
QG3(7)	2	515 (0, 2)	2	515 (0, 2)	2	515 (0, 2)	2	514 (0, 2)
QG4(7)	2	291 (0, 1)	2	278 (0, 1)	2	278 (0, 1)	2	269 (0, 1)
QG5(7)	1	168 (0, 0)	1	71 (0, 0)	1	71 (0, 0)	1	60 (0, 0)
QG6(7)	3	257 (0, 1)	4	257 (0, 1)	4	257 (0, 1)	2	257 (0, 1)
QG7	3	263 (0, 1)	4	259 (0, 1)	4	259 (0, 1)	2	258 (0, 1)
Allsquares(37)	49	385 (0, 5)	162	229 (0, 4)	33	278 (0, 4)	4	271 (0, 4)
AllsquaresUnsat(37)	49	745 (0, 15)	160	683 (0, 13)	33	728 (0, 13)	4	660 (0, 13)
Bibd1011(6)	19	5 (0, 0)	30	3 (0, 0)	15	6 (0, 0)	9	6 (0, 0)
Bibd1213(7)	28	13 (0, 0)	42	20 (0, 0)	20	15 (0, 0)	7	2 (0, 0)
Bibd6(10)	5	1 (0, 0)	8	1 (0, 0)	4	1 (0, 0)	3	1 (0, 0)
Bibd7(14)	6	1 (0, 0)	9	1 (0, 0)	5	1 (0, 0)	4	1 (0, 0)
Bibd8(7)	9	14 (0, 0)	14	3 (0, 0)	7	11 (0, 0)	4	2 (0, 0)
Bibd9(10)	13	3 (0, 0)	20	2 (0, 0)	9	3 (0, 0)	6	4 (0, 0)
BibdVariousK(29)	17	343 (0, 4)	23	298 (0, 4)	14	324 (0, 5)	6	266 (0, 3)
bqwh15106_glb(10)	0	0 (0, 0)	1	0 (0, 0)	1	0 (0, 0)	1	0 (0, 0)
bqwh18141_glb(10)	1	0 (0, 0)	1	0 (0, 0)	1	0 (0, 0)	1	0 (0, 0)
Cjss(10)	61	1084 (0, 6)	97	1085 (0, 6)	86	1091 (0, 6)	24	944 (0, 5)
Compet02(20)	199	67 (0, 0)	1165	200 (2, 2)	1164	200 (2, 2)	22	9 (0, 0)
Compet08(16)	17	146 (0, 1)	90	16 (0, 0)	91	16 (0, 0)	73	463 (0, 4)
CostasArray(11)	3	577 (0, 3)	15	381 (0, 2)	16	514 (0, 3)	2	362 (0, 2)
LatinSquare(10)	1	180 (0, 1)	2	180 (0, 1)	2	180 (0, 1)	1	180 (0, 1)
MagicSquare(18)	1057	1179 (10, 11)	1208	1103 (11, 11)	1444	1400 (14, 14)	629	756 (6, 7)
Medium(5)	305	117 (0, 0)	1717	1446 (4, 4)	1721	1446 (4, 4)	31	10 (0, 0)
Nengfa(3)	113	19 (0, 0)	777	5 (0, 0)	770	5 (0, 0)	4	6 (0, 0)
pigeons_glb(19)	0	0 (0, 0)	0	0 (0, 0)	0	0 (0, 0)	1	0 (0, 0)
PseudoGLB(100)	122	466 (5, 23)	142	482 (7, 26)	12	382 (0, 18)	95	488 (5, 26)
Rcpsp(39)	0	0 (0, 0)	1	0 (0, 0)	0	0 (0, 0)	1	0 (0, 0)
RcpspTighter(39)	0	0 (0, 0)	1	0 (0, 0)	0	0 (0, 0)	1	0 (0, 0)
Small(5)	14	2 (0, 0)	87	1 (0, 0)	87	1 (0, 0)	4	1 (0, 0)
Total	210	412 (55, 114)	163	273 (24, 78)	124	274 (20, 75)	44	252 (11, 70)

Table 1. Experiments comparing different encodings with sugar.

ferent ASP encodings as well as *sugar* (2.2.1). The name of the benchmark class and the number of instances is given in the first column. In each setting, the *trans* column shows the average time used for translating CSP problems into their final propositional format. For this purpose, *aspartame* uses *gringo* (4.5), while *sugar* uses a dedicated implementation resulting in a CNF in DIMACS format. Analogously, the *solve* column gives the average time for each benchmark class, showing the number of translation (to^t) and total timeouts (*to*). In all cases, we use *clasp* (3.1.1) as back-end ASP or SAT solver, respectively, in its ASP default configuration `tweety`. Comparing the *basic encoding* with the *optimized encoding*, we observe that the latter significantly reduces both solving and grounding timeouts (mainly due to the *CabinetStart1* class). Next, we want to investigate the impact of the recursive structure of our encodings. For this, we disabled splitting of linear constraints within *sugar*'s translation. This usually leads to

an exponential increase in the number of clauses for *sugar*. The results are given in column *optimizednosplit* of Table1. In fact, disabled splitting performs as good as the optimized encoding with splitting. In some cases, it even improves performance. As splitting constraints the right way usually depends heavily on heuristics, our recursive translation offers a heuristic-independent solution to this problem. Finally, although *aspartame* and *sugar* are at eye height regarding solving time and timeouts, *aspartame* falls short by an order of magnitude when it comes to translating CSPs into propositional format. Here the dedicated implementation of *sugar*[8] clearly outperforms the grounder-based approach of *aspartame*. On the other hand, our declarative approach allows us to easily modify and thus experiment with different encodings.

This flexibility was extremely useful when elaborating upon different encodings. While for the benchmarks used in Table 1 the *alldifferent* constraints are translated to linear constraints with the help of *sugar*, we now want to handle them by an encoding. To this end, Table 2 compares four alternative encodings for handling *alldifferent*.

	alldiffA		*alldiffB*		*alldiffC*		*alldiffD*		*alldiffBnosplit*	
Benchmark Class	trans	solve(to^t,to)	trans	solve(to^t,to)	trans	solve(to^t,to)	trans	solve(to^t,to)	trans	solve(to^t,to)
CabinetStart1(40)	54	20(0, 0)	53	20(0, 0)	53	20(0, 0)	65	23(0, 0)	8	2(0, 0)
QG3(7)	2	515(0, 2)	2	515(0, 2)	2	515(0, 2)	3	515(0, 2)	2	515(0, 2)
QG4(7)	2	276(0, 1)	2	278(0, 1)	2	283(0, 1)	3	290(0, 1)	2	278(0, 1)
QG5(7)	1	56(0, 0)	1	68(0, 0)	1	60(0, 0)	3	149(0, 0)	1	68(0, 0)
QG6(7)	4	257(0, 1)	4	257(0, 1)	4	257(0, 1)	6	258(0, 1)	4	257(0, 1)
QG7	4	259(0, 1)	4	260(0, 1)	4	259(0, 1)	4	261(0, 1)	4	260(0, 1)
Allsquares(37)	161	229(0, 4)	161	230(0, 4)	161	230(0, 4)	166	232(0, 4)	33	281(0, 4)
AllsquaresUnsat(37)	158	683(0,13)	158	682(0,13)	158	682(0,13)	168	701(0,13)	32	726(0,13)
Bibd1011(6)	30	3(0, 0)	30	3(0, 0)	31	3(0, 0)	32	3(0, 0)	15	6(0, 0)
Bibd1213(7)	42	20(0, 0)	42	20(0, 0)	43	20(0, 0)	43	24(0, 0)	20	15(0, 0)
Bibd6(10)	8	1(0, 0)	8	1(0, 0)	8	1(0, 0)	14	1(0, 0)	4	1(0, 0)
Bibd7(14)	9	1(0, 0)	9	1(0, 0)	9	1(0, 0)	9	1(0, 0)	5	1(0, 0)
Bibd8(7)	14	3(0, 0)	14	3(0, 0)	14	3(0, 0)	14	3(0, 0)	7	11(0, 0)
Bibd9(10)	20	2(0, 0)	20	2(0, 0)	20	2(0, 0)	21	3(0, 0)	9	3(0, 0)
BibdVariousK(29)	23	298(0, 4)	23	298(0, 4)	23	298(0, 4)	26	300(0, 4)	14	324(0, 5)
bqwh15106_glb(10)	0	0(0, 0)	0	0(0, 0)	0	0(0, 0)	4	1(0, 0)	0	0(0, 0)
bqwh18141_glb(10)	0	0(0, 0)	0	0(0, 0)	0	0(0, 0)	5	1(0, 0)	0	0(0, 0)
Cjss(10)	99	1085(0, 6)	99	1085(0, 6)	99	1085(0, 6)	97	1085(0, 6)	87	1091(0, 6)
Compet02(20)	834	22(0, 0)	836	18(0, 0)	840	21(0, 0)	1095	268(2, 2)	842	19(0, 0)
Compet08(16)	236	463(0, 4)	232	455(0, 4)	230	475(0, 4)	633	1498(0,13)	233	455(0, 4)
CostasArray(11)	5	503(0, 2)	5	349(0, 2)	5	494(0, 3)	10	517(0, 3)	6	343(0, 2)
LatinSquare(10)	0	180(0, 1)	0	180(0, 1)	0	180(0, 1)	2	180(0, 1)	0	180(0, 1)
MagicSquare(18)	1192	1107(11, 11)	1192	1103(11, 11)	1191	1104(11, 11)	1196	1106(11, 11)	1442	1401(14, 14)
Medium(5)	1510	36(0, 0)	1499	34(0, 0)	1503	35(0, 0)	1700	1458(4, 4)	1507	34(0, 0)
Nengfa(3)	10	2(0, 0)	9	1(0, 0)	9	2(0, 0)	67	117(0, 0)	9	1(0, 0)
pigeons_glb(19)	0	0(0, 0)	0	0(0, 0)	0	0(0, 0)	0	0(0, 0)	0	0(0, 0)
PseudoGLB(100)	142	482(7,26)	142	482(7,26)	142	482(7,26)	142	483(7,26)	12	382(0,18)
Rcpsp(39)	1	0(0, 0)	1	0(0, 0)	1	0(0, 0)	1	0(0, 0)	0	0(0, 0)
RcpspTighter(39)	1	0(0, 0)	1	0(0, 0)	1	0(0, 0)	1	0(0, 0)	0	0(0, 0)
Small(5)	62	1(0, 0)	62	1(0, 0)	61	1(0, 0)	92	8(0, 0)	62	1(0, 0)
Total	148	269(18,76)	148	266(18,76)	148	269(18,77)	174	326(24,92)	110	264(14,72)

Table 2. Experiments comparing different encodings for alldifferent.

While variants *A*, *B*, *C* have already been presented above, variant *D* uses a more complex encoding using hall intervals of size three [11]. Our experiments show however that

[8] The timeouts of *sugar* during translation are always due to insufficient memory.

simple translations using binary inequalities like *A* and *B* are as good as more complex ones like *C* and even outperform more sophisticated ones as *D*. The last column shows the combination of non-splitting linear constraints (in *sugar*) and handling the *alldifferent* constraint with translation *B*. This is currently the best performing combination of encodings and constitutes the default setting of *aspartame* (2.0.0)[9].

5 Discussion

CASP approaches [13–16] handle constraints in a lazy way by using off-the-shelf CP solvers either as back-end (*ezcsp*) or online propagator (*clingcon*). The approach in [15] (inca) uses a dedicated propagator to translate constraints (via the order encoding) during solving. Our approach can be seen as an "early" approach, translating all constraints before the solving process. As regards pure CP solving, *aspartame*'s approach can be seen as a first-order alternative to SAT-based approaches like *sugar* [5]. Although the performance of the underlying SAT solver is crucial, the SAT encoding plays an equally important role [17]. Among them, we find the direct [18, 19], support [20, 21], log [22, 23], order [4, 5], and compact order [24] encoding. The order encoding showed good performance for a wide range of CSPs [4, 12, 25–27]. In fact, the SAT-based CP solver *sugar* won the `GLOBAL` category at the 2008 and 2009 CP solver competitions [28]. Also, the SAT-based CP solver BEE [29] and the CLP system B-Prolog [30] use this encoding. In fact, the order encoding provides a compact translation of arithmetic constraints, while also maintaining bounds consistency by unit propagation.

We presented an alternative approach to solving finite linear CSPs based on ASP. The resulting system *aspartame* relies on high-level ASP encodings and delegates both the grounding and solving tasks to general-purpose ASP systems. Furthermore, these encodings can be used as a library for solving CSPs as part of an encompassing logic program, as it is done in the framework of CASP. We have contrasted *aspartame* with its SAT-based ancestor *sugar*, which delegates only the solving task to off-the-shelf SAT solvers, while using dedicated algorithms for constraint preprocessing. Although *aspartame* does not fully match the performance of *sugar* from a global perspective, the picture is fragmented and leaves room for further improvements, especially for the translation process. Experience from *aspartame* will definitely help to build/improve the current experimental support for linear constraints in *gringo*. Despite all this, *aspartame* demonstrates that ASP's general-purpose technology allows to compete with state-of-the-art constraint solving techniques. In fact, the high-level approach of ASP facilitates extensions and variations of first-order encodings for dealing with particular types of constraints. In the future, we thus aim at investigating alternative encodings, e.g., regarding *alldifferent* constraints, as well as support of further global constraints.

References

1. Banbara, M., Gebser, M., Inoue, K., Schaub, T., Soh, T., Tamura, N., Weise, M.: Aspartame: Solving CSPs with ASP. In ASPOCP. abs/1312.6113, CoRR (2013)

[9] The system is available at http://www.cs.uni-potsdam.de/aspartame/

2. Rossi, F., v. Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier (2006)
3. Biere, A., Heule, M., v. Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS (2009)
4. Crawford, J., Baker, A.: Experimental results on the application of satisfiability algorithms to scheduling problems. In AAAI. AAAI Press (1994) 1092–1097
5. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints **14** (2009) 254–272
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
7. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012)
8. Beldiceanu, N., Simonis, H.: A constraint seeker: Finding and ranking global constraints from examples. In CP. Springer (2011) 12–26
9. Tamura, N., Banbara, M., Soh, T.: Compiling pseudo-boolean constraints to SAT with order encoding. In ICTAI. IEEE (2013) 1020–1027
10. Gent, I., Nightingale, P.: A new encoding of alldifferent into SAT. In Workshop on Modelling and Reformulating Constraint Satisfaction Problems. (2004)
11. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C., Walsh, T.: Decompositions of all different, global cardinality and related constraints. In IJCAI. (2009) 419–424
12. Soh, T., Inoue, K., Tamura, N., Banbara, M., Nabeshima, H.: A SAT-based method for solving the two-dimensional strip packing problem. Fund. Informaticae **102** (2010) 467–487
13. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In ICLP. Springer (2009) 235–249
14. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In ASPOCP (2009) 16–30
15. Drescher, C., Walsh, T.: A translational approach to constraint answer set solving. Theory and Practice of Logic Programming **10** (2010) 465–480
16. Ostrowski, M., Schaub, T.: ASP modulo CSP: The clingcon system. Theory and Practice of Logic Programming **12** (2012) 485–503
17. Prestwich, S.: CNF encodings. In [3]. 75–97
18. de Kleer, J.: A comparison of ATMS and CSP techniques. In IJCAI. (1989) 290–296
19. Walsh, T.: SAT v CSP. In CP. (2000) 441–456
20. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence **45** (1990) 275–286
21. Gent, I.: Arc consistency in SAT. In ECAI. (2002) 121–125
22. Iwama, K., Miyazaki, S.: SAT-variable complexity of hard combinatorial problems. In IFIP. (1994) 253–258
23. Van Gelder, A.: Another look at graph coloring via propositional satisfiability. Discrete Applied Mathematics **156** (2008) 230–243
24. Tanjo, T., Tamura, N., Banbara, M.: Azucar: A SAT-based CSP solver using compact order encoding (tool presentation). In SAT. (2012) 456–462
25. Metodi, A., Codish, M., Stuckey, P.: Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. J. of Artificial Intell. Research **46** (2013) 303–341
26. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14** (2009) 357–391
27. Banbara, M., Matsunaka, H., Tamura, N., Inoue, K.: Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In LPAR. (2010) 112–126
28. Lecoutre, C., Roussel, O., van Dongen, M.: Promoting robust black-box solvers through competitions. Constraints **15** (2010) 317–326
29. Metodi, A., Codish, M.: Compiling finite domain constraints to SAT with BEE. Theory and Practice of Logic Programming **12** (2012) 465–483
30. Zhou, N.: The SAT compiler in B-prolog. The ALP Newsletter, March 2013 (2013)