clingo goes Linear Constraints over Reals and Integers

T. Janhunen[1], R. Kaminski[3], M. Ostrowski[3], T. Schaub[23], S. Schellhorn[3], and P. Wanko[3]

[1]Aalto University [2]INRIA Rennes [3]University of Potsdam

Abstract. The recent series 5 of the ASP system *clingo* provides generic means to enhance basic Answer Set Programming (ASP) with theory reasoning capabilities. We instantiate this framework with different forms of linear constraints and elaborate upon its formal properties. Given this, we discuss the respective implementations, and present techniques for using these constraints in a reactive context. More precisely, we introduce extensions to *clingo* with difference and linear constraints over integers and reals, respectively, and realize them in complementary ways. Finally, we empirically evaluate the resulting *clingo* derivatives *clingo*[DL] and *clingo*[LP] on common language fragments and contrast them to related ASP systems.

1 Introduction

Answer Set Programming (ASP; [3]) has become an established paradigm for knowledge representation and reasoning, in particular, when it comes to solving knowledge-intense combinatorial (optimization) problems. Despite its versatility, however, ASP falls short in handling non-Boolean constraints, such as linear constraints over reals or unlimited integers. This shortcoming was broadly addressed in the recent *clingo* 5 series [12] by providing generic means for incorporating theory reasoning. They span from theory grammars for seamlessly extending *clingo*'s input language with theory expressions to a simple interface for integrating theory propagators into *clingo*'s solver component.

We instantiate this framework with different forms of linear constraints and elaborate upon its formal properties. Given this, we discuss the respective implementations, and present techniques for using these constraints in a reactive context. In more detail, we introduce extensions to *clingo* with difference and linear constraints over integers and reals, respectively, and realize them in complementary ways. For handling difference constraints, we provide customized implementations of well-established algorithms in Python and C++, while we use *clingo*'s Python API to connect to off-the-shelf linear programming solvers, viz. *cplex* and *lpsolve*, to deal with linear constraints. In both settings, we support integer as well as real valued variables. For a complement, we also consider *clingcon*, a derivative of *clingo*, integrating constraint propagators for handling linear constraints over integers at a low-level. While this fine integration must be done at compile-time, the aforementioned Python extensions are added at run-time. Our empirical analysis complements the study in [18] with experimental results on our new systems *clingo*[DL] and *clingo*[LP]. Finally, we provide a comparison of different semantic options for integrating theories into ASP and a systematic overview of the various features of state-of-the-art ASP systems handling linear constraints.

2 Answer Set Programming with Linear Constraints

Our paper centers upon the theory reasoning capabilities of *clingo* that allow us to extend ASP with linear constraints, also referred to as ASP[LC]. We focus below on the corresponding syntactic and semantic features, and refer the reader to the literature for an introduction to the basics of ASP.

We consider (disjunctive) *logic programs with linear constraints*, for short[1] *lc-programs*, over sets $\mathcal{A}$ and $\mathcal{L}$ of ground *regular* and *linear constraint atoms*, respectively. An expression is said to be *ground*, if it contains no ASP variables. Accordingly, such programs consist of *rules* of the form

$$\texttt{a}_1\texttt{;}\dots\texttt{;a}_m \texttt{ :- } \texttt{a}_{m+1}\texttt{,}\dots\texttt{,a}_n\texttt{,not a}_{n+1}\texttt{,}\dots\texttt{,not a}_o$$

where each $\texttt{a}_i$ is either a regular atom in $\mathcal{A}$ of form $\texttt{p(t}_1\texttt{,...,t}_k\texttt{)}$ such that all $\texttt{t}_i$ are ground terms or an lc-atom in $\mathcal{L}$ of form[2] '$\texttt{\&sum\{a}_1\texttt{*x}_1\texttt{;...;a}_l\texttt{*x}_l\texttt{\}<=k}$' that stands for the linear constraint $a_1 \cdot x_1 + \dots + a_l \cdot x_l \leq k$. All $\texttt{a}_i$ and $\texttt{k}$ are finite sequences of digits with at most one dot[3] and represent real-valued coefficients a_i and k. Similarly all $\texttt{x}_i$ stand for the real (or integer) valued variables x_i. As usual, `not` denotes (default) *negation*. A rule is called a *fact* if $m, o = 1$, *normal* if $m = 1$, and an *integrity constraint* if $m = 0$. A linear constraint of form $x_1 - x_2 \leq k$ is called a *difference constraint*, and represented as '$\texttt{\&sum\{x}_1\texttt{; -1*x}_2\texttt{\}<=k}$' (or '$\texttt{\&diff\{x}_1\texttt{-x}_2\texttt{\}<=k}$' in pure difference logic settings).

To ease the use of ASP in practice, several extensions have been developed. First of all, rules with ASP variables are viewed as shorthands for the set of their ground instances. Further language constructs include *conditional literals* and *cardinality constraints* [21]. The former are of the form $\texttt{a:b}_1\texttt{,...,b}_m$, the latter can be written as $\texttt{s\{d}_1\texttt{;...;d}_n\texttt{\}t}$, where `a` and $\texttt{b}_i$ are possibly default-negated (regular) literals and each $\texttt{d}_j$ is a conditional literal; `s` and `t` provide optional lower and upper bounds on the number of satisfied literals in the cardinality constraint. We refer to $\texttt{b}_1\texttt{,...,b}_m$ as a *condition*, and call it *static* if it is evaluated during grounding, otherwise it is called *dynamic*. The practical value of such constructs becomes apparent when used with ASP variables. For instance, a conditional literal like `a(X):b(X)` in a rule's antecedent expands to the conjunction of all instances of `a(X)` for which the corresponding instance of `b(X)` holds. Similarly, `2{a(X):b(X)}4` is true whenever at least two and at most four instances of `a(X)` (subject to `b(X)`) are true.

Likewise, *clingo*'s syntax of linear constraints offers several convenience features. As above, elements in linear constraint atoms can be conditioned (and use ASP variables), viz. '$\texttt{\&sum\{a}_1\texttt{*x}_1\texttt{:c}_1\texttt{;...;a}_l\texttt{*x}_l\texttt{:c}_n\texttt{\}<=k}$' where each $\texttt{c}_i$ is a condition. As above, the usage of ASP variables allows for forming arbitrarily long expressions (cf. Listing 1.2). That is, by using static or dynamic conditions, we may formulate linear constraints that are determined relative to a problem instance during grounding and even dynamically during solving, respectively. Also, linear constraints can be formed with further relations, viz. >=, <, >, =, and !=. Moreover, the theory language for linear constraints offers a domain declaration for real variables, '`&dom{lb..ub}=x`' expressing that all values of

[1] We keep using the prefix '*lc*-' throughout as a shorthand for concepts related to linear constraints.

[2] In *clingo*, theory atoms are preceded by '&'.

[3] In the input language of *clingo*, sequences containing dots must be quoted to avoid clashes.

`x` must lie between `lb` and `ub`, inclusive. And finally the maximization (or minimization) of an objective function can be expressed with `&maximize{`$\mathtt{a}_1$`*`$\mathtt{x}_1$`:`$\mathtt{c}_1$`;...;`$\mathtt{a}_l$`*`$\mathtt{x}_l$`:`$\mathtt{c}_n$`}` (or by `minimize`). The full theory grammar for linear constraints over reals is available at `https://potassco.org/clingo/examples`.

Semantically, a logic program induces a set of *stable models*, being distinguished models of the program determined by the stable models semantics [15]. To extend this concept to logic programs with linear constraints, we follow the approach of lazy theory solving [4]. We abstract from the specific semantics of a theory by considering the lc-atoms representing the underlying linear constraints. The idea is that a regular stable model X of a program over $\mathcal{A} \cup \mathcal{L}$ is only valid wrt the theory, if the constraints induced by the truth assignment to the lc-atoms in $\mathcal{L}$ are satisfiable in the theory. In our setting, this amounts to finding an assignment of reals (or integers) to all numeric variables that satisfies a set of linear constraints induced by $X \cap \mathcal{L}$. Although this can be done in several ways, as detailed below, let us illustrate this by a simple example. The (non-ground) logic program containing the fact '`a("1.5").`' along with the rule '`&sum{R*x}<=7 :- a(R).`' has the regular stable model {`a("1.5")`, `&sum{"1.5"*x}<=7`}. Here, we easily find an assignment, e.g. $\{x \mapsto 4.2\}$, that satisfies the only associated linear constraint '$1.5 * x \leq 7$'.

In what follows, we make this precise by instantiating the general framework of logic programs with theories in [12] to the case of linear constraints over reals and integers (and so difference constraints). Also, we focus on one theory at a time. Thereby, our emphasis lies on the elaboration of alternative semantic options for stable models with linear constraints, which pave the way for different implementation techniques discussed in Section 4.

We use the following notation. Given a rule r as above, we call $\{\mathtt{a}_1, \dots, \mathtt{a}_m\}$ its *head* and denote it by $H(r)$. Furthermore, we define $H(P) = \bigcup_{r \in P} H(r)$.

First of all, we may distinguish whether linear constraints are only determined outside or additionally inside a program. Accordingly, we partition $\mathcal{L}$ into *defined* and *external* lc-atoms, namely $\mathcal{L} \cap H(P)$ and $\mathcal{L} \setminus H(P)$, respectively.[4] Thus, while external lc-atoms must only be satisfied by the respective theory, defined ones must additionally be derivable through rules in the program. The second distinction is about the logical correspondence between theory atoms and theory constraints. To this end, we partition $\mathcal{L}$ into *strict* and *non-strict* lc-atoms, denoted by $\mathcal{L}^{\leftrightarrow}$ and $\mathcal{L}^{\rightarrow}$. The strict correspondence requires a linear constraint to be satisfied *iff* the associated lc-atom in $\mathcal{L}^{\leftrightarrow}$ is true. A weaker condition is imposed in the non-strict case. Here, a linear constraint must hold *only if* the associated lc-atom in $\mathcal{L}^{\rightarrow}$ is true. Thus, only lc-atoms in $\mathcal{L}^{\rightarrow}$ assigned true impose requirements, while constraints associated with falsified lc-atoms in $\mathcal{L}^{\rightarrow}$ are free to hold or not. However, by contraposition, a violated constraint leads to a false lc-atom.

Different combinations of such correspondences are possible, and we may even treat some constraints differently than others. In view of this, we next provide an extended definition of stable models that accommodates all above correspondences. Following [12], we accomplish this by mapping the semantics of lc-programs back to regular stable models. To this end, we abstract from the actual constraints and identify a solution with a

[4] This distinction is analogous to that between head and input atoms, defined via rules or `#external` directives [13], respectively.

set of linear constraint atoms. More precisely, we call $\mathcal{S} \subseteq \mathcal{L}$ a *linear constraint solution*, if there is an assignment of reals (or integers) to all real (integer) valued variables represented in $\mathcal{L}$ that

(i) satisfies all linear constraints associated with strict and non-strict lc-atoms in $\mathcal{S}$ and
(ii) falsifies all linear constraints associated with strict lc-atoms in $\mathcal{L}^{\leftrightarrow} \setminus \mathcal{S}$.

Then, we define a set $X \subseteq \mathcal{A} \cup \mathcal{L}$ as an *lc-stable model* of an lc-program P, if there is some lc-solution $\mathcal{S} \subseteq \mathcal{L}$ such that X is a (regular) stable model of the logic program

$$P \cup \{\texttt{a.} \mid \texttt{a} \in (\mathcal{L}^{\leftrightarrow} \setminus H(P)) \cap \mathcal{S}\} \cup \{\texttt{:- not a.} \mid \texttt{a} \in (\mathcal{L}^{\leftrightarrow} \cap H(P)) \cap \mathcal{S}\} \quad (1)$$
$$\cup \{\texttt{\{a\}.} \mid \texttt{a} \in (\mathcal{L}^{\rightarrow} \setminus H(P)) \cap \mathcal{S}\} \cup \{\texttt{:- a.} \mid \texttt{a} \in (\mathcal{L} \cap H(P)) \setminus \mathcal{S}\}. \quad (2)$$

The rules added to P express conditions aligning the lc-atoms in $X \cap \mathcal{L}$ with a corresponding lc-solution $\mathcal{S}$. To begin with, the set of facts on the left in (1) makes sure that all lc-atoms in $\mathcal{S}$ that are external and strict also belong to X. Unlike this, the corresponding set of choice rules in (2) merely says that non-strict external lc-atoms from $\mathcal{S}$ may be included in X or not. The integrity constraints in (1) and (2) take care of defined lc-atoms, viz. the ones in $H(P)$. The respective set in (1) again focuses on strict lc-atoms and stipulates the ones from $\mathcal{S}$ are included in X as well. Finally, for both strict and non-strict defined lc-atoms, the integrity constraints in (2) assert the falsity of atoms that are not in $\mathcal{S}$.

In what follows, we elaborate upon the formal relationships among the different types of lc-atoms. To this end, we distinguish four homogeneous settings, in which all lc-atoms are either defined+strict, defined+non-strict, external+strict, or external+non-strict, respectively. We use the following notation. For an lc-program P over $\mathcal{A} \cup \mathcal{L}$ and an lc-solution $\mathcal{S} \subseteq \mathcal{L}$, we define $P|_{\mathcal{S}}$ as the extension of program P given in (1) and (2). Also, let $\mathcal{X}(P)$ denote the set of (regular) stable models of program P over $\mathcal{A} \cup \mathcal{L}$, and $\mathcal{X}_{\text{LC}}(P) = \bigcup_{\mathcal{S} \subseteq \mathcal{L} \text{ lc-solution}} \mathcal{X}(P|_{\mathcal{S}})$ its set of lc-stable models. Note that the respective semantic setting is determined by the type of lc-atoms in $\mathcal{L}$. In fact, two syntactically equivalent lc-programs may yield different lc-models in different settings. This is made precise in the following propositions.

Theorem 1. *Let P be an lc-program over $\mathcal{A} \cup \mathcal{L}$ and P' an lc-program over $\mathcal{A} \cup \mathcal{L}'$ such that $P = P'$.*

1. *If $\mathcal{L} = \mathcal{L} \cap H(P)$, then $\mathcal{X}_{LC}(P) \subseteq \mathcal{X}(P)$*
2. *If $\mathcal{L} = \mathcal{L}^{\rightarrow} \setminus H(P)$, then $\mathcal{X}(P) \subseteq \mathcal{X}_{LC}(P)$*
3. *If $\mathcal{L}' = \mathcal{L}'^{\rightarrow}$, then $\mathcal{X}_{LC}(P) \subseteq \mathcal{X}_{LC}(P')$*

Note that $P = P'$ also makes $\mathcal{L}$ and $\mathcal{L}'$ syntactically equivalent, although they may represent different types of lc-atoms. The above results draw on the observation that if all atoms in $\mathcal{L}'$ are non-strict, then $\{\mathcal{S} \subseteq \mathcal{L} \mid \mathcal{S} \text{ is an lc-solution}\} \subseteq \{\mathcal{S} \subseteq \mathcal{L}'^{\rightarrow} \mid \mathcal{S} \text{ is an lc-solution}\}$. This is because the former set of lc-solutions at least need to satisfy condition (i) while the latter only have to satisfy (i). Note that Proposition 1 does not just apply to ASP[LC] but to ASP modulo arbitrary theories.

In more detail, Proposition 1.1 expresses that each lc-stable model is also a regular stable model in a setting involving defined lc-atoms only. Conversely, Proposition 1.2

expresses that each regular stable model is also an lc-stable model in the external+non-strict setting. Proposition 1.3 portrays that handling lc-atoms in a strict or non-strict way may lead to fewer (or equal) lc-stable models than treating them just in a non-strict way.

In contrast to the observations of Proposition 1, the following proposition tells us that regular and lc-stable models are in general incomparable in the external+strict setting.

Theorem 2. *There exist lc-programs P over $\mathcal{A} \cup \mathcal{L}$ with $\mathcal{L} = \mathcal{L}^{\leftrightarrow} \setminus H(P)$, so that $\mathcal{X}(P) \nsubseteq \mathcal{X}_{LC}(P)$ or $\mathcal{X}_{LC}(P) \nsubseteq \mathcal{X}(P)$.*

This results from the fact that the treatment of strict lc-atoms may prune regular stable models and, on the other hand, the pure external evaluation of lc-atoms may induce additional stable models.

Now that we have explored the formal correspondence among the alternative settings, let us discuss their appropriateness for ASP[LC]. To this end, let us consider two examples.

We first asses the two defined settings. Modifying our above example, let P_1 be

```
{a("1.5")}.  &sum{"1.5"*x}<=7 :- a("1.5").
&sum{x}<"4.5".
```

along with its two regular stable models $X_1 = \{$ `&sum{x}<"4.5"` $\}$ and $X_2 = \{$ `a("1.5"), &sum{"1.5"*x}<=7, &sum{x}<"4.5"` $\}$.

Let us first consider the defined+strict case, in which the lc-atoms `&sum{"1.5"*x}<=7` and `&sum{x}<"4.5"` belong to $\mathcal{L}^{\leftrightarrow} \cap H(P)$. Then, $\mathcal{S}_a = \emptyset$ is an lc-solution, since both $1.5 * x \leq 7$ and $x < 4.5$ can be falsified. However, the resulting program $P_1|_{\mathcal{S}_a}$ contains rules '`&sum{x}<"4.5".`' and '`:- &sum{x}<"4.5".`' and thus admits no regular stable model. The same result is obtained for $\mathcal{S}_b = \{$ `&sum{"1.5"*x}<=7` $\}$. Unlike this, $\mathcal{S}_c = \{$ `&sum{x}<"4.5"` $\}$ is no lc-solution although it appears to support X_1 as an lc-model. In a strict setting, an *iff* correspondence is imposed between lc-atoms and their associated linear constraints. This excludes $\mathcal{S}_c$ as an lc-solution, since there is no real-valued assignment satisfying $x < 4.5$ while falsifying $1.5 * x \leq 7$. This situation is caused by the non-derivabality of lc-atom `&sum{"1.5"*x}<=7`, which is in turn falsified by the stable models semantics. The strict interpretation of the lc-atom then requires the falsification of $1.5 * x \leq 7$. Finally, $\mathcal{S}_d = \{$ `&sum{x}<"4.5", &sum{"1.5"*x}<=7` $\}$ is another lc-solution. Given that $P_1|_{\mathcal{S}_d} = P_1 \cup \{$ `:- not &sum{"1.5"*x}<=7. :- not &sum{x}<"4.5".` $\}$ has the regular stable model X_2, we establish that X_2 is the only lc-stable model of P_1.

This example has illustrated that strict lc-atoms impose a rather strong connection to their associated constraints in a defined setting. Hence, let us consider next the above example in a defined+non-strict setting, requiring merely an *only if* condition between constraints and their lc-atoms. Now, $\mathcal{S}_c = \{$ `&sum{x}<"4.5"` $\}$ is an lc-solution since $1.5 * x \leq 7$ must not be falsified. Accordingly, the regular stable model X_1 of $P_1|_{\mathcal{S}_c} = P_1 \cup \{$ `:- &sum{"1.5"*x}<=7.` $\}$ attests that X_1 is also an lc-stable model of P_1. The other lc-solutions yield the same results as above.

Next, let us analyze the two external settings. For this, let the lc-program P_2 be

```
:- not &sum{x}<"4.5".  a("1.5") :- &sum{"1.5"*x}<=7.
```

admitting no regular stable models, due to the included integrity constraint.

First, we examine the external+non-strict setting. In this case, each combination of the lc-atoms `&sum{"1.5"*x}<=7` and `&sum{x}<"4.5"` in $\mathcal{L}^{\rightarrow} \setminus H(P)$ results in an lc-solution. However, the existence of lc-stable models depends upon the presence of lc-atom `&sum{x}<"4.5"`. Lc-models are obtained if it is included, otherwise the integrity constraint in P_2 denies them. The lc-solution $\mathcal{S}_a = \{$ `&sum{x}<"4.5"` $\}$ results in the identical lc-stable model. Note that all underlying assignments must satisfy $x < 4.5$ and hence $1.5 * x \leq 7$. However, the non-strict nature of `&sum{"1.5"*x}<=7` leaves its truth value open. Thus, stable model semantics sets it to false and `a("1.5")` is not obtained although the actual constraint $1.5 * x \leq 7$ in the rule body in P_2 is satisfied. Similarly, the lc-solution $\mathcal{S}_b = \{$ `&sum{x}<"4.5", &sum{"1.5"*x}<=7` $\}$ induces the same counter-intuitive lc-model $\{$ `&sum{x}<"4.5"` $\}$ along with a second, arguably more intuitive lc-model $\{$ `a("1.5"), &sum{"1.5"*x}<=7, &sum{x}<"4.5"` $\}$.

The previous discussion has revealed that non-strict lc-atoms may ignore information induced by the theory in an external setting. This lack is compensated in an external+strict setting by the above condition (ii) and the resulting assertion of lc-atoms representing satisfied constraints in (1). Accordingly, $\{$ `a("1.5"), &sum{"1.5"*x}<=7, &sum{x}<"4.5"` $\}$ is the only lc-stable model of P_2. By interpreting both lc-atoms in a strict manner, the inclusion of `&sum{x}<"4.5"` entails that of `&sum{"1.5"*x}<=7` as well. Hence, the singleton $\{$ `&sum{x}<"4.5"` $\}$ cannot be an lc-model of P_2 in a external+strict setting.

The previous discussion has shown that certain semantic combinations are more appropriate for treating linear constraints than others. This may be different for other theories. We have seen that a defined+strict interpretation of lc-atoms may be overly strong, since the non-derivability of lc-atoms may severely restrict real-valued assignments. Conversely, the external+non-strict treatment of lc-atoms may be too weak, since it admits real-valued variable assignments satisfying constraints that are not reflected in the corresponding lc-stable models. As a consequence, we focus in what follows on the external+strict and defined+non-strict settings for lc-atoms.

3 Multi-Shot ASP Solving with Linear Constraints

Multi-shot solving [13] is about solving continuously changing logic programs in an operative way. This can be controlled via reactive procedures that loop on solving while reacting, for instance, to outside changes or previous solving results. These reactions may entail the addition or retraction of rules that the operative approach can accommodate by leaving the unaffected program parts intact within the solver. This avoids re-grounding and benefits from heuristic scores and nogoods learned over time. In fact, evolving logic programs with linear constraints can be extremely useful in dynamic applications, for example, to add new resources in a planning domain, or to set the value of an observed variable measured using sensors. The abstraction from actual constraints to constraint atoms allows us to easily extend multi-shot solving to lc-programs.

To illustrate how seamlessly our systems *clingo*[DL] and *clingo*[LP] support multi-shot solving, let us apply the exemplary Python script, shipped with *clingo* to illustrate incremental solving, to model the spoiling Yale shooting scenario [6]. Multi-shot solving in *clingo* relies on two directives (cf. [13]), the `#program` directive for regrouping rules and the `#external` directive for declaring atoms as being external to the program at hand. The truth value of such external atoms is set via *clingo*'s API. The aforementioned Python script loops over increasing integers until a stop criterion is met. It presupposes three groups of rules declared via `#program` directives. At step 0 the programs named `base` and `check(n)` are grounded and solved for `n` = 0. Then, in turn programs `check(n)` and `step(n)` are added for `n` > 0, grounded, and the resulting overall program solved. In addition, at each step `n` an external atom `query(n)` is introduced; it is set to true for the current iteration `n` and false for all previous instances with smaller integers than `n`. We refer the reader to [13] for further details on the Python part. Notably, for dealing with lc-programs, we can use the exemplary Python script as is—once the respective propagator is registered with the solver.

In the spoiled Yale shooting scenario [6], we have a gun and two actions, load and shoot. If we load, the gun becomes loaded. If we shoot, it kills the turkey, if the gun was loaded for no more than 35 minutes. Otherwise, the gun powder is spoiled. We model this planning problem in ASP[LC]. We start by including the incremental Python program,

```
1  #include "incmode_lc.lp".

3  #program base.
4    action(load).       action(shoot).      action(wait).
5  duration(load,25).  duration(shoot,5).  duration(wait,36).
6  unloaded(0).
7  &sum { at(0) } = 0.
8  &sum { armed(0) } = 0.
```

Listing 1.1. Spoiled Yale shooting instance

the grammar, and the propagator for linear constraints in the first line of Listing 1.1.[5] This listing is the base program. All actions and their durations are introduced in Lines 4 and 5. At the initial situation, the gun is unloaded (Line 6). Line 7 and 8 initialize integer variables `at(0)` and `armed(0)` with 0 (see below). Listing 1.2 gives the dynamic part of the problem; it is grounded for each step `n`. Line 2 enforces that exactly one action is done per step. The exact times at which each step takes place is captured by the integer variables `at(n)`. The difference between two consecutive time steps is the duration of the respective action (Line 3). The next three lines make the fluents inertial, viz. the gun stays loaded/unloaded if it was loaded/unloaded before, and the turkey stays dead. Lines 9 and 10 use the integer variable `armed(n)` to describe for how long the weapon has been loaded at step `n`. Whenever it is unloaded, `armed(n)` is 0, otherwise it is increased by the duration of the last action. The upcoming four lines (12–15) encode the conditions and effects of the actions. When we load the gun, it becomes loaded; when we shoot, it becomes unloaded. If we shoot and the gun was loaded for no longer than 35 minutes (and thus the gun powder is unspoiled), then the turkey is dead. The last line ensures that we cannot shoot if the gun is not loaded. Together with the initial situation and the actions from Listing 1.1 this encodes the spoiled Yale shooting problem, and any solution represents an executable plan. Listing 1.3 adds a query to our problem.

[5] For uniformity, we use semi-colons '`;`' rather than '`,`' for separating body elements.

```
#program step(n).
1{do(X,n) : action(X)}1.
&sum { at(n); -1*at(N') } = D :- do(X,n); duration(X,D); N'=n-1.

loaded(n)   :- loaded(n-1); not unloaded(n).
unloaded(n) :- unloaded(n-1); not loaded(n).
dead(n)     :- dead(n-1).

&sum { armed(n) } = 0 :- unloaded(n-1).
&sum { armed(n); -1*armed(N') } = D :- do(X,n); duration(X,D); N'=n-1; loaded(N').

loaded(n)   :- do(load,n).
unloaded(n) :- do(shoot,n).
dead(n)     :- do(shoot,n); &sum { armed(n) } <= 35.
:- do(shoot,n); unloaded(n-1).
```

Listing 1.2. Spoiled Yale shooting scenario

```
1 #program check(n).
2 :- not dead(n); query(n).
3 :- not &sum { at(n) } <= 100; query(n).
4 :- do(shoot,n); not &sum { at(n) } > 35.
```

Listing 1.3. Query for the spoiled Yale Shooting Scenario.

In Line 2 we require that the turkey is dead at step `n`. As this constraint is subject to the external atom `query(n)`, it is only active at solving step `n`. The next line ensures that we kill the turkey within 100 minutes. And as an additional constraint, we added some preparation time such that we are not allowed to shoot in the first 35 minutes. It is possible to solve this problem within three steps. There exist two solutions at this time point, one of them containing `unloaded(0)`, `do(wait,1)`, `unloaded(1)`, `do(load,2)`, `loaded(2)`, `do(shoot,3)`, `unloaded(3)`, `dead(3)`. That is, we simply wait before loading and shooting. The second solution loads the gun instead of waiting, thus loading the gun twice before shooting.

4 *clingo* derivatives and related systems

In this section, we give an overview of systems extending ASP with linear constraints. We start with our own systems *clingo*[DL] and *clingo*[LP] both relying upon *clingo*'s interface for theory propagators. We also include *clingcon*, since it is based on a much lower level API using the internal functions of *clingo* (and *clasp*) in C++. While *clingcon* implements a highly sophisticated system using advanced preprocessing and solving techniques, the Python variants of *clingo*[DL] and *clingo*[LP] provide easily modifiable and maintainable propagators for difference and linear constraints, respectively. This carries over to the C++ variant of *clingo*[DL] since the C++ and Python API share the same functionality. Table 1 shows a comparative list of features for these systems. The

Table 1. Feature comparison

	Python	C++	strict	non-strict	external	defined	n-ary	reals	optimization
clingo[DL]	✓	✓	✓[1]	✓	✓	✓	✗	✓[2]	✓[3]
clingo[LP]	✓	✗	✓	✓	✓	✓	✓	✓	✓[4]
clingcon	✗	✓	✓	✗	✓	✗[5]	✓	✗	✓

[1] Only with Python API
[2] Only for non-strict lc-atoms
[3] Needs an additional plugin
[4] Optimization is relative to stable models
[5] Theory atoms in rule heads are shifted into negative body

two flexible *clingo* derivatives support all four combinations of strict/non-strict and defined/external lc-atom types, whereas *clingcon* has a fixed one. Also the bandwidth of supported constraints is different. While *clingo*[DL] only supports difference constraints, the other two support n-ary linear constraints. *clingo*[DL] and *clingo*[LP] support (approximations of) real numbers (see below). Moreover, all three *clingo* derivatives allow for optimizing objective functions over the numeric variables (in addition to optimization in ASP).

***clingo*[DL]** extends *clingo* with difference constraints of the form $x - y \leq k$, where x and y are integer (or real) variables and k is an integer (real) constant. Despite the restriction to two variables, they allow for naturally encoding timing related problems, as e.g., in scheduling, and are solvable in polynomial time. *clingo*[DL] uses *clingo*'s theory interface to realize a stateful propagator that checks during search whether the current set of implied difference constraints is satisfiable [8]. To this end, it makes use of the stateful nature of the theory interface that allows for incrementally updating internal states and thus for backtracking to previous states without having to rebuild the internal representation. By default, all difference constraint atoms are considered to be non-strict. In this case, it is only necessary to keep track of lc-atoms that are assigned true since only then the constraint is required to hold. In the strict case, false assignments to difference constraint atoms are considered as well. This is done by adding $y - x \leq -k - 1$ whenever '`&diff{x-y}<=k`' is assigned false. As a side-product of the satisfiability check, an integer (real) assignment for all variables is obtained and ultimately printed for all lc-stable models. Usually, several or even an infinite number of assignments exist. The returned assignment is the one with the lowest sum of the absolute values of all variables. For instance, in terms of scheduling problems, this amounts to scheduling each job as soon as possible.

***clingo*[LP]** fully covers the extension of ASP[LC] described in Section 2. This *clingo* derivative accepts lc-atoms containing integer and real variables possibly subject to dynamic conditions. That is, *clingo*[LP] extends ASP with constraints as dealt with in Linear Programming (LP; [10]) as well as according objective functions for optimization. In *clingo*[LP], the latter takes all linear constraints induced by the Boolean assignment into account. As above, the theory interface of *clingo* is used to integrate a stateful propagator that checks during search the satisfiability of the current set of linear constraints. Here, however, this is done with a generic interface to dedicated LP solvers, currently supporting *cplex* and *lpsolve*. (Note that both LP solvers do an exponential consistency check.) The Python interfaces of *cplex* and *lpsolve* natively support relations $=$, $\geq$, and $\leq$. We add support for $<$, $>$, and $\neq$. To this end, we translate $<$ and $>$ into $\leq$ and $\geq$ by subtracting or adding an ε to the right-hand-side of a linear constraint, respectively.[6] Furthermore, $\neq$ is treated as a disjunction of $<$ and $>$. By default, *clingo*[LP] treats lc-atoms in a non-strict manner. Thus only linear constraints represented by true lc-atoms are considered. When treating them strictly, false lc-atoms are handled using the complementary relation. In this case, the corresponding linear constraint is derived by using the complementary relation. Notably, *clingo*[LP] offers dynamic conditions in lc-atoms. This allows for linear constraints of variable length even during search. All conditions have to be decided before such a constraint is included in the consistency check. Furthermore, *clingo*[LP] is able to update its internal state incrementally but rebuilds the linear constraint system after backtracking to avoid accumulating rounding errors. Also, it uses an Irreducible Inconsistent Set algorithm [24] for extracting minimal sets of conflicting constraints to support conflict learning in the ASP solver. On the one hand, this extraction is expensive, on the other hand, such core conflicts may significantly reduce the search space. To control this trade-off, *clingo*[LP] only enables this feature after a certain percentage of lc-atoms and conditions is assigned (by default 20%). Similarly, frequent theory consistency checks are expensive and a conflict is less likely to be found within a small assignment; accordingly, an analogous percentage based threshold allows for controlling their invocation (default 0%).

clingcon series 3 offers a *clingo*-based ASP system with constraints over integers [2]; it is implemented in C++ and features a strict, external semantics. Sophisticated preprocessing techniques are supported and non-linear constraints such as the global distinct constraint are translated into linear ones. Integer variables are represented using the order encoding [9], and customized propagators using state-of-the-art lazy nogood and variable generation are employed. The propagators do not only ensure bound consistency on the variables but also derive new bounds. Furthermore, multi-objective optimization on the integer variables is supported. In contrast to *clingo*[LP], conditions on integer variables must be static.

All systems are available at `https://potassco.org/{clingoDL,clingoLP, clingcon}`.

Big picture. Finally, let us relate our systems with others extending ASP with linear constraints. The first category, referred to as translation-based approaches, includes systems such as *ezsmt* [18], *dingo* [17], *aspmt2smt* [5], and *mingo* [19]. The first three translate both ASP and constraints into SAT Modulo Theories (SMT; [4]); *dingo* is restricted to difference constraints. Unlike this, *mingo*'s target formalism is Mixed Integer Linear Programming (MILP). Furthermore, *aspartame* [1] translates ASP[LC] (over integers) back to ASP by using the order encoding. Once the input program is translated, only a solver for the target formalism is needed. This is one of the advantages of translation-based approaches. Also, they benefit from the features and performance of the respective target systems. A drawback is the translation itself since it may result in large propositional representations or weak propagation strength. The second category extends the standard Conflict Driven Nogood Learning (CDNL; [14]) machinery of

[6] This ε can be configured using the command line and defaults to 10^{-3} (as in *cplex*).

ASP solvers with constraint propagators. This allows for propagating both Boolean and linear constraints during search. The latter is thus continuously checked for consistency and even new constraints may get derived. For instance, the *clingo*-based system *dlvhex*[CP] [20] uses *gecode*, while *ezcsp* uses a Prolog constraint solver for consistency checking. Unlike this, *inca* [11] extends a previous *clingo* version with a customized lazy propagator generating constraints according to the order encoding. This approach allows for deriving new constraints such as bounds of the integer variables.

The *clingo* derivatives *clingo*[DL] and *clingo*[LP] belong to the second category of systems, just like *clingcon* 3. Table 2 summarizes important similarities and differences

Table 2. Comparing related applications

	clingo [DL]	*clingo* [LP]	*clingcon*	*aspartame*	*inca*	*ezcsp*	*ezsmt*	*mingo*	*dingo*	*aspmt2smt*	*dlvhex* [CP]
translation	✗	✗	✓[1]	✓	✗	✗	✓	✓	✓	✓	✗
explicit	✗	✗	✓[2]	✓	✓	✗	✗	✗	✗	✗	✗
non-linear	✗[3]	✗	✓[4]	✓[4]	✓	✓	✓	✗	✗[3]	✓	✓
real numbers	✗	✓	✗	✗	✗	✓	✓	✓[5]	✗	✓	✗
optimization	✗	✓[6]	✓	✓	✓	✗	✗	✗	✗	✗	✓
non-tight	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓

[1] Allows for partial problem translations
[2] Lazily created
[3] Only difference constraints
[4] Translation of distinct into linear constraints
[5] Only for variables
[6] Optimization relative to stable models

of the aforementioned systems. The first row tells us whether a system relies on a translation to SMT, MILP, or ASP. The second one indicates whether the approach uses some form of explicit variable representation. This is the case when using an encoding and usually results in a large number of propositional atoms to represent variables with large domains. Half of the systems are able to handle constraints over reals while the other half is restricted to integers. Note that for a system of inequalities, a solution over reals can be found much easier than one over integers. For all systems, real numbers are implemented as floating point numbers. Due to this, round-off errors cannot completely be avoided. Note that since computers are finite precision machines, the imprecision of floating point computations is common to any computer systems and/or languages [16]. *cplex* uses numerically stable methods to perform its linear algebra so that round-off errors usually do not cause problems.[7] With "non-linear" we distinguish systems handling global or non-linear constraints, and "non-tight" indicates whether a system can deal with recursive programs. Finally, the table lists all systems that are able to optimize an objective function over the integer and/or real variables.

[7] See *Numeric difficulties* at `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.7.0/ilog.odms.studio.help/pdf/usrcplex.pdf`

5 Experimental analysis

We begin with an empirical analysis of our *clingo* derivatives in different settings. We investigate, first, different types of lc-atoms, viz. defined+non-strict versus external+strict, second, different levels of theory interfaces, Python or C++, for *clingo*[DL], and, third, different levels of integration, namely, dedicated implementations versus off-the-shelf solver. Finally, we contrast the performance of our systems with other systems for ASP[LC].

We ran each benchmark on a Xeon E5520 2.4 GHz processor under Linux limiting RAM to 20 GB and execution time to 1800s. For *clingo*[DL] and *clingo*[LP], we use *clingo* 5.2.0. Furthermore, we use *clingcon* 3.2.0, *dingo* v.2011-09-23, *mingo* v.2012-09-30, *ezsmt* 1.0.0, and *ezcsp* 1.7.9 for our experiments. We upgraded *dingo* and *mingo* to use recent versions of their back-end solvers. Hence, in our experiments, the LP-based systems *clingo*[LP] and *mingo* use *cplex* 12.7.0.0 and the SMT-based systems *dingo* and *ezsmt* use *z3* 4.4.2. The benchmark set consists of 165 instances, among which 110 can be encoded using difference constraints (DL) and 55 require linear constraints with more than two variables (LC). In detail, the DL set consists of 38 instances of two-dimensional strip packing (2SP) [22], and 72 instances of flow shop (FS), job shop (JS), and open shop (OS) problems [23], selecting three instances for each job and machine at random. Since not all systems support optimization over variable values, we bounded the instances with 1.2 times the best known bound and solved the resulting decision problem. The LC instance set includes 20 instances of incremental scheduling (IS), 15 instances of reverse folding (RF), and 20 instances of weighted sequence (WS). Encodings have been adopted from [18] in combination with the instances from the ASP competition.[8] Our empirical evaluation focuses on available systems sharing comparable encodings. This was not the case for *aspartame*, *aspmt2smt*, *inca*, and *dlvhex*[CP]. The first two systems have a proper and thus different input language and encoding philosophy, *inca* produced incorrect results (cf. [2] for details), and *dlvhex*[CP] is no longer maintained.

Table 3 compares *clingo*[DL] and *clingo*[LP] with different encoding techniques, types of theory atoms, and programming language hosting the theory interface by measuring average time (T) and timeouts (TO). Each column consists of one combination of form

Table 3. Comparison of *clingo* derivatives *clingo*[DL] and *clingo*[LP]

		DL/DNS/PY		DL/ES/PY		LP/DNS/PY		LP/ES/PY		DL/DNS/CPP		DL/ES/CPP	
CLASS	#inst	T	TO	T	TO	T	TO	T	TO	T	TO	T	TO
2SP	38	344	6	484	9	1346	23	1753	36	**148**	**3**	342	7
FS	35	678	11	1541	27	1221	21	1800	35	**465**	**5**	1349	26
JS	24	1261	15	1229	14	1800	24	1800	24	**534**	**4**	678	7
OS	13	8	**0**	17	**0**	963	6	1532	10	**0**	**0**	**0**	**0**
DL	110	611	32	928	50	1360	74	1752	105	**316**	**12**	695	40

[8] We refrained from using the other three benchmark classes from this source because the available instances were too easy in view of producing informative results.

system/*atom*/*language*, where *system* is either DL or LP for *clingo*[DL] and *clingo*[LP], *atom* either DNS or ES for defined+non-strict and external+strict lc-atoms, and *language* either PY or CPP for Python and C++, respectively. To compare *clingo*[DL] and *clingo*[LP], we restrict the set of benchmarks to DL. We observe that DNS performs better than ES in all settings. Under lc-stable model semantics, defined lc-atoms are more tightly constrained. External lc-atoms, on the other hand, induce an implicit choice leading to a larger search space and might introduce duplicate solutions with different assignments. Furthermore, strict lc-atoms double the amount of implications that have to be considered by the propagator. As expected, the C++ variant of *clingo*[DL] outperforms its Python counterpart, even though the performance gain does not reach an order of magnitude.

Table 4 compares different systems dealing with ASP[LC] by average time (T) and timeouts (TO). Only the best configurations from Table 3 were selected for comparison.

Table 4. Comparison of different systems for ASP with linear constraints

		DL/DNS/CPP		LP/DNS/PY		*clingcon*		*dingo*		*mingo*		*ezsmt*		*ezcsp*	
CLASS	#inst	T	TO	T	TO	T	TO	T	TO	T	TO	T	TO	T	TO
2SP	38	148	3	1346	23	**3**	**0**	403	7	292	5	318	6	1800	38
FS	35	**465**	**5**	1221	21	1022	19	1047	20	1040	16	1667	32	735	9
JS	24	534	4	1800	24	**277**	**3**	1258	15	1423	18	1315	15	1800	24
OS	13	**0**	**0**	963	6	1	**0**	4	**0**	76	**0**	24	**0**	1044	7
DL	110	**316**	**12**	1360	74	387	22	765	42	743	39	930	52	1372	78
IS	20	–	–	1800	20	**582**	**5**	–	–	649	7	648	7	1620	18
RF	15	–	–	1680	14	**21**	**0**	–	–	542	1	121	**0**	1013	7
WS	20	–	–	1800	20	27	**0**	–	–	90	**0**	**12**	**0**	1800	20
LC	55	–	–	1767	54	**227**	**5**	–	–	416	8	273	7	1520	45
all	165	–	–	1564	128	**307**	**27**	–	–	580	47	602	59	1446	123

All systems were tested using their default configurations. For DL, DL/DNS/CPP performs best overall, even though *clingcon* is better for 2SP and JS. The class FS generates the most difference constraints among all benchmark classes, making it less suited for translation-based approaches, like *dingo*, *mingo*, and *ezsmt*, and producing overhead for more involved propagation as in *clingcon*. By default, *ezcsp* performs the theory consistency check on full answer sets, and by doing so avoids handling vast amounts of constraints during search and therefore performs comparatively well on FS. For the other classes though, this generate and test approach is less effective. Regarding LC and overall results, *clingcon* clearly dominates the competition, followed by the two translation-based approaches *mingo* and *ezsmt* with underlying state-of-the-art solvers *cplex* and *z3*, respectively. LP/DNS/PY falls behind, since it is a straightforward Python implementation and uses an exponential consistency check. In addition, distinct features of *clingo*[LP] like real-valued variables and optimization as well as dynamic conditions are not supported by other systems and thus not included in the benchmark set.

6 Summary

We presented several truly hybrid ASP systems incorporating difference and linear constraints. Previous approaches addressed this by resorting to translations into foreign solving paradigms like MILP or SMT. This difference is analogous to the one between genuine ASP solvers like *clasp* and *wasp* and earlier ones like *assat* and *cmodels* that translate ASP to SAT. The resulting systems *clingo*[DL] and *clingo*[LP] comprise several complementary aspects. For instance, *clingo*[DL] relies upon customized propagators, one variant using a Python API, the other a C++ API. This is similar to the approach of *inca* and *clingcon* 3 for Constraint ASP. Unlike this, *clingo*[LP] builds upon the Python API to incorporate off-the-shelf LP solvers for propagation, optionally *cplex* or *lpsolve*. This is similar to the approach of *dlvhex*[CP] and *clingcon* 2 integrating *gecode* for constraint processing. Both *clingo*[DL] and *clingo*[LP] allow for dealing with integer as well as real variables. The former admits two, the latter an arbitrary number of such variables per linear constraint. This is complemented by *clingcon* 3 adding constraint processing to *clingo* by using a low level API.

We accomplished this by instantiating the generic framework of ASP modulo theories described in [12]. We defined lc-stable models and elaborated upon different types of lc-atoms, ultimately settling on the combinations defined+non-strict and external+strict for *clingo*[DL] and *clingo*[LP].[9] Our underlying formal analysis on the interaction of strict- and definedness has actually a much broader impact given that other systems follow similar principles. Although we lack a deeper analysis, *inca* and *dlvhex*[CP] appear to adhere to an external+strict treatment of constraint atoms, just as our previous systems *clingcon*, *dingo*, and *mingo*, while *ezsmt* and *ezcsp* seem to follow an external+non-strict approach. Moreover, the results in Proposition 1 are of a general nature and apply well beyond ASP systems dealing with linear constraints.

We provided a conceptual and empirical comparison of *clingo*[DL] and *clingo*[LP] with related systems for dealing with different forms of linear constraints in ASP. Our experiments focused on, first, examining different types of lc-atoms and APIs for both *clingo* derivatives, and, second, comparing them with related systems. In the first case, *clingo*[DL] using defined+non-strict lc-atoms along with the C++ API yields the best results, and in the second one, the aforementioned *clingo*[DL] configuration outperforms the other systems for the set of benchmarks only involving difference constraints, and *clingcon* has an edge over all other systems regarding the set of benchmarks featuring arbitrary (integer-based) linear constraints.

Finally, we showed how easily our machinery can be applied to online reasoning scenarios by using *clingo*‘s multi-shot and theory reasoning capabilities in tandem.

References

1. M. Banbara, M. Gebser, K. Inoue, M. Ostrowski, A. Peano, T. Schaub, T. Soh, N. Tamura, and M. Weise. aspartame: Solving constraint satisfaction problems with answer set programming. In F. Calimeri, G. Ianni, and M. Truszczyński, editors, *Proceedings of the Thirteenth*

[9] This is our recommendation in view of our analysis in Section 2; both systems actually support all four combinations of strict/non-strict and defined/external lc-atoms.

International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15), volume 9345 of *Lecture Notes in Artificial Intelligence*, pages 112–126. Springer-Verlag, 2015.

2. M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub. Clingcon: The next generation. *Theory and Practice of Logic Programming*, 2017. To appear.
3. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
4. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.
5. M. Bartholomew and J. Lee. System aspmt2smt: Computing ASPMT theories by SMT solvers. In E. Fermé and J. Leite, editors, *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*, volume 8761 of *Lecture Notes in Artificial Intelligence*, pages 529–542. Springer-Verlag, 2014.
6. P. Cabalar, R. Otero, and S. Pose. Temporal constraint networks in action. In W. Horn, editor, *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI'00)*, pages 543–547. IOS Press, 2000.
7. M. Carro and A. King, editors. *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16)*, volume 52. Open Access Series in Informatics (OASIcs), 2016.
8. S. Cotton and O. Maler. Fast and flexible difference constraint propagation for DPLL (T). In A. Biere and C. Gomes, editors, *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 170–183. Springer-Verlag, 2006.
9. J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In B. Hayes-Roth and R. Korf, editors, *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 1092–1097. AAAI Press, 1994.
10. G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
11. C. Drescher and T. Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming*, 10(4-6):465–480, 2010.
12. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. Theory solving made easy with clingo 5. In Carro and King [7], pages 2:1–2:15.
13. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume arXiv:1405.3694v1 of *Theory and Practice of Logic Programming, Online Supplement*, 2014. Available at `http://arxiv.org/abs/1405.3694v1`.
14. M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.
15. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
16. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.
17. T. Janhunen, G. Liu, and I. Niemelä. Tight integration of non-ground answer set programming and satisfiability modulo theories. In P. Cabalar, D. Mitchell, D. Pearce, and E. Ternovska, editors, *Proceedings of the First Workshop on Grounding and Transformation for Theories with Variables (GTTV'11)*, pages 1–13, 2011.
18. Y. Lierler and B. Susman. SMT-based constraint answer set solver EZSMT (system description). In Carro and King [7], pages 1:1–1:15.

19. G. Liu, T. Janhunen, and I. Niemelä. Answer set programming via mixed integer programming. In G. Brewka, T. Eiter, and S. McIlraith, editors, *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, pages 32–42. AAAI Press, 2012.
20. A. De Rosis, T. Eiter, C. Redl, and F. Ricca. Constraint answer set programming based on HEX-programs. In *Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015), August 31, 2015, Cork, Ireland*, August 2015.
21. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
22. T. Soh, K. Inoue, N. Tamura, M. Banbara, and H. Nabeshima. A SAT-based method for solving the two-dimensional strip packing problem. *Fundamenta Informaticae*, 102(3-4):467–487, 2010.
23. E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.
24. J. van Loon. Irreducibly inconsistent systems of linear inequalities. In *European Journal of Operational Research*, volume 3, pages 283–288. Elsevier Science, 1981.