Solving Configuration Problems with ASP and Declarative Domain-Specific Heuristics[1]

Richard Taupe[2] and **Gerhard Friedrich**[3] and **Konstantin Schekotihin**[3] and **Antonius Weinzierl**[4]

Abstract. Domain-specific heuristics are an essential technique for solving configuration problems efficiently. Current approaches to integrate domain-specific heuristics with Answer Set Programming (ASP) are unsatisfactory when dealing with heuristics that are specified non-monotonically on the basis of partial assignments. Such heuristics frequently occur in practice, for example, when picking a component that has not yet been placed in a configuration problem. Therefore, we present novel syntax and semantics for declarative specifications of domain-specific heuristics in ASP. Our approach supports heuristic statements that depend on the partial assignment maintained during solving, which has not been possible before. We provide an implementation in ALPHA that makes ALPHA the first lazy-grounding ASP system to support declaratively specified domain-specific heuristics. Two well-known configuration problems are used to demonstrate the benefits of our proposal. The experiments confirm that combining lazy-grounding ASP solving and our novel heuristics can be vital for solving industrial-size configuration problems.

1 INTRODUCTION

Answer Set Programming (ASP) [2, 14, 20, 25] is a declarative programming approach that has successfully been applied to product configuration [11, 12, 19, 21, 27].

However, large-scale industrial problems are challenging for ASP. One issue is the so-called *grounding bottleneck*: Large problem instances often cannot be grounded by modern grounders like GRINGO [16] or I-DLV [4] in adequate time and space [8]. Another issue is that, even if the problem can be grounded, computation of answer sets may take considerable time, as indicated by the ASP Competitions [5, 18].

Lazy grounding is a technique that tackles the grounding bottleneck. The approach presented in this paper is implemented in the lazy-grounding system ALPHA [35]. Thus we can handle large-scale problem instances.

The main topic of this work concerns *domain-specific heuristics*, an essential technique for improving solving performance by equipping the solver with hints on how to solve a problem efficiently. Several approaches to integrate domain-specific heuristics with ASP solving have already been proposed: a procedural approach for the WASP system [7] and a declarative approach for CLINGO [17].

Both existing approaches to integrate domain-specific heuristics with ASP solving are unsatisfactory: Procedural heuristics counteract the declarative nature of ASP, and the existing declarative approach does not permit dynamic heuristics reasoning about partial assignments. Dynamically evaluating heuristics w.r.t. a partial assignment can be an essential feature for practical domains. For example, heuristics in product configuration may need to compute the amount of space left after placing a component.

We tackle the challenge of finding a satisfying solution to integrate declarative domain-specific heuristics with ASP programs. To this end, we extend CLINGO's approach. Our extension supports dynamic heuristics while at the same time keeping the language simple and easy to use.

We implemented our approach as an extension to the lazy-grounding ASP system ALPHA. Two well-known configuration problems – the House Reconfiguration Problem (HRP) and the Partner Units Problem (PUP) – are used to demonstrate our approach and to obtain experimental results. The experiments confirm that combining lazy-grounding ASP solving and our novel heuristics can be vital for solving industrial-size configuration problems.

After briefly covering preliminaries in Section 2, we introduce the state of the art of domain-specific heuristics in ASP in Section 3. Our novel approach is presented in Section 4. Section 5 describes applications and experimental results, and Section 6 concludes this paper.

2 PRELIMINARIES

The declarative programming approach of ASP allows a programmer to formulate the problem as a logic program specifying the search space and the properties of solutions instead of stating how to solve a problem. An ASP solver then finds models (so-called *answer sets*) for this logic program, which correspond to solutions for the original problem.

Most state-of-the-art ASP systems implement "ground and solve", i.e., they first produce the full grounding (i.e., variable-free equivalent) of the input program, which is then solved. This results in the so-called *grounding bottleneck*: As soon as the grounding exceeds all available memory, the problem cannot be solved anymore. *Lazy grounding* avoids the grounding bottleneck by interleaving grounding and solving. This approach is implemented by ASP systems such as ALPHA [24, 35].

Due to space constraints, we assume familiarity with syntax and semantics of ASP, and refer to [3, 13, 27] for details.

In the remainder of this section, we introduce the notation that will

[1] This paper contains extracts from a journal paper [28] that extends an earlier conference paper [29] and is currently under review.

[2] Siemens AG Österreich and Alpen-Adria-Universität Klagenfurt, Austria, email: richard.taupe@siemens.com (Richard Taupe is the main author of this paper; other authors contributed equally and are listed in the alphabetical order.)

[3] Alpen-Adria-Universität Klagenfurt, Austria

[4] TU Wien, Vienna, Austria

be used later in the article.

An *assignment* A is a set of signed literals $\mathbf{T}a$, $\mathbf{F}a$, or $\mathbf{M}a$, where $\mathbf{T}a$ and $\mathbf{F}a$ express that an atom a is true and false, respectively, and $\mathbf{M}a$ indicates that a "must-be-true". $\mathbf{M}$ signifies that an atom must eventually become true by derivation in a correct solution extending the current partial assignment, but no derivation has yet been found that would make the atom true. Let $A_s = \{a \mid s\ a \in A\}$ for $s \in \{\mathbf{F}, \mathbf{M}, \mathbf{T}\}$ denote the set of atoms occurring with a specific sign in assignment A. We assume assignments to be *consistent*, i.e., no negative literal may also occur positively ($A_\mathbf{F} \cap (A_\mathbf{M} \cup A_\mathbf{T}) = \emptyset$), and every positive literal must also occur with must-be-true ($A_\mathbf{T} \subseteq A_\mathbf{M}$).

An assignment A is *complete* if every atom is assigned true or false. An assignment that is not complete is *partial*.

The function truth_A for a (partial) assignment A maps an atom to the truth value that the atom is currently assigned in the given assignment, or to $\mathbf{U}$ if the atom is currently unassigned:

$$\mathsf{truth}_A(a) = \begin{cases} \mathbf{F} & a \in A_\mathbf{F}, \\ \mathbf{M} & a \in A_\mathbf{M} \setminus A_\mathbf{T}, \\ \mathbf{T} & a \in A_\mathbf{T}, \\ \mathbf{U} & \text{otherwise.} \end{cases}$$

3 DOMAIN-SPECIFIC HEURISTICS IN ASP

State-of-the-art ASP solvers are well suited to solve a wide range of problems, as shown in ASP competitions, experiments, and (industrial) applications reported in the literature [9, 11, 18, 23]. However, domain-specific heuristics are needed to achieve breakthroughs in solving industrial configuration problems with ASP. Several approaches have implemented embedding heuristic knowledge into the ASP solving process.

HWASP [7] facilitates external procedural heuristics that are consulted at specific points during the solving process via an API. As a result, HWASP can find solutions for all published instances of the Partner Units Problem (PUP) by exploiting external heuristics formulated in C++ or Python.

The CLINGO system supports a declarative approach to formulating domain-specific heuristics in ASP in the form of `#heuristic` directives [13, 17]. Heuristics extend the ASP language to allow for declarative specification of atom weights and signs for the solver's internal heuristics. An atom's weight influences the order in which atoms are considered by the solver when making a decision. A sign modifier instructs whether the selected atom must be assigned true or false. Atoms with a higher weight are assigned a value before atoms with a lower weight.

CLINGO's semantics for heuristic directives seem to introduce some limitations for formulating heuristics that require to reason about the absence of truth-values for atoms, e.g. the absence of decisions in a search state. For example, in configuring technical systems, we might prefer to assign, in the current search state, the most relevant yet unplaced electronic component to a free slot of a motherboard. These limitations are discussed in detail in our other publications [28, 29].

To overcome this issue we propose, in the following section, to evaluate negation as failure (i.e., not) in heuristic statements *w.r.t. the current partial assignment* in the solver. This partial assignment represents the search state. As a consequence, not X is true if X is false *or unassigned* in a partial assignment.

4 DYNAMIC DECLARATIVE HEURISTICS

Declaratively specifying domain-specific heuristics in ASP plays a vital role in enabling ASP to solve large-scale industrial problems. CLINGO has been the only ASP system to support such heuristics so far. Although language and semantics of heuristic directives in CLINGO have shown to be beneficial in many cases, dynamic aspects of negation as failure in heuristic conditions have not been addressed satisfactorily.

We present novel syntax and semantics for heuristic directives in ASP that improve this situation. We assume that the underlying solver can assign one of three values to any atom: *true* (denoted with $\mathbf{T}$), *false* ($\mathbf{F}$), and *must-be-true* ($\mathbf{M}$) (cf. [35]). The following definitions can be used without modification for solvers that do not use the third truth value $\mathbf{M}$. The set of atoms assigned must-be-true will be empty in this case.

Definition 1 (Heuristic Directive) *A* heuristic directive *is of the form ⟨1⟩, where ha_i ($0 \leq i \leq n$) are heuristic atoms of the form $s_i\ a_i$, in which $s_0 \in \{\{\mathbf{F}\}, \{\mathbf{T}\}\}$ and $s_i \subseteq \{\mathbf{F}, \mathbf{M}, \mathbf{T}\}$ are sets of sign symbols and a_i is an atom, and w and l are integer terms.*

$$\texttt{\#heuristic}\ ha_0 : ha_1, \ldots, ha_k, \text{not}\ ha_{k+1}, \ldots, \text{not}\ ha_n. \quad [w@l] \qquad \langle 1 \rangle$$

The heuristics' head is given by ha_0 and its condition by $\{ha_1, \ldots, ha_k, \text{not}\ ha_{k+1}, \ldots, \text{not}\ ha_n\}$, which is similar to a rule body.

Where the meaning is clear from the context, we may omit all symbols except sign symbols themselves in a set of sign symbols, e.g., we write $\mathbf{TM}$ instead of $\{\mathbf{T}, \mathbf{M}\}$.

Example 1 *Consider the following heuristic directive h:*

$$\texttt{\#heuristic}\ \mathbf{F}\mathrm{a} : \mathbf{TM}\mathrm{b}, \mathbf{T}\mathrm{c}, \text{not}\ \mathbf{TMF}\mathrm{d}.$$

This directive means that the atom a *shall be assigned* $\mathbf{F}$ *if* b *is assigned* $\mathbf{T}$ *or* $\mathbf{M}$, c *is assigned* $\mathbf{T}$, *and* d *is not assigned.*

We now introduce some notation that will be used in further definitions. The function atm maps a heuristic atom ha_i to a_i by removing the sign, and a set of heuristic atoms to the set of atoms occurring in them (e.g., $\mathsf{atm}(\mathbf{M}\mathrm{a}) = \mathrm{a}$, $\mathsf{atm}(\{\mathbf{M}\mathrm{a}, \mathbf{T}\mathrm{b}\}) = \{\mathrm{a}, \mathrm{b}\}$). The function signs maps a heuristic atom ha_i to s_i by removing the atom (e.g., $\mathsf{signs}(\mathbf{M}\mathrm{a}) = \{\mathbf{M}\}$).

The *head* of a heuristic directive h of the form ⟨1⟩ is denoted by $\mathsf{head}(h) = ha_0$, its *weight* by $\mathsf{weight}(h) = w$ if given, else 0, and its *level* by $\mathsf{level}(h) = l$ if given, else 0. The *(heuristic) condition* of a heuristic directive h is denoted by $\mathsf{cond}(h) := \{ha_1, \ldots, ha_k, \text{not}\ ha_{k+1}, \ldots, \text{not}\ ha_n\}$, the *positive condition* is $\mathsf{cond}^+(h) := \{ha_1, \ldots, ha_k\}$ and the *negative condition* is $\mathsf{cond}^-(h) := \{ha_{k+1}, \ldots, ha_n\}$.

Let HA be a set of heuristic atoms. Then, for $s \subseteq \{\mathbf{F}, \mathbf{M}, \mathbf{T}\}$, $HA|_s = \{a \mid s\ a \in HA\}$ denotes the set of atoms in HA whose set of sign symbols equals s.

Example 2 *Consider the heuristic directive h from Example 1 again. Here,* $\mathsf{cond}^+(h)|_\mathbf{MT} = \{\mathrm{b}\}$; $\mathsf{cond}^+(h)|_\mathbf{T} = \{\mathrm{c}\}$; *and* $\mathsf{cond}^+(h)|_\mathbf{F} = \emptyset$. *Furthermore,* $\mathsf{cond}^-(h)|_\mathbf{FMT} = \{\mathrm{d}\}$; *note that the order of sign symbols does not matter due to set semantics.*

The distinguishing features of our approach are as follows:

- Each heuristic atom contains a set of sign symbols. Each sign symbol represents one of the truth values $\mathbf{F}$ (*false*), $\mathbf{T}$ (*true*), and $\mathbf{M}$ (*must-be-true*).
- In the condition, sign symbols provide a richer way of controlling when the condition is satisfied. A positive literal in the condition is satisfied if the truth value currently assigned to its atom is contained in its set of sign symbols (which is $\{\mathbf{M}, \mathbf{T}\}$ by default if not explicitly given). A negative literal in the condition is satisfied if the truth value currently assigned to its atom is *not* contained in its set of sign symbols or if its atom is currently not assigned any truth value.
- In the heuristic head, the sign symbol is used to determine the truth value to be chosen by the heuristics. If s_0 is $\mathbf{T}$ or empty, the heuristics makes the solver guess a_0 to be true; if s_0 is $\mathbf{F}$, a_0 will be made false.[5]
- Terms w and l denote weight and level as familiar from optimize-statements in ASP-Core-2 [3] or weak constraints in DLV [22]. The level is more important than the weight; both default to 0, and together they are called priority.

We now describe our semantics more formally, beginning with the condition under which a heuristic atom is satisfied.

Definition 2 (Satisfying a Heuristic Atom) *Given a ground heuristic atom ha and a partial assignment A, ha is* satisfied *w.r.t. A iff:* $\mathsf{truth}_A(\mathsf{atm}(ha)) \in \mathsf{signs}(ha)$, *i.e., if its atom is assigned a truth value that is included in the heuristic atom's sign set.*[6]

Whether a heuristic directive is satisfied depends on whether the atoms occurring in the directive are satisfied.

Definition 3 (Satisfying a Heuristic Directive) *Given a ground heuristic directive h and a partial assignment A,* $\mathsf{cond}(h)$ *is* satisfied *w.r.t. A iff: every $ha \in \mathsf{cond}^+(h)$ is satisfied and no $ha \in \mathsf{cond}^-(ha)$ is satisfied.*

Intuitively, a heuristic condition is satisfied iff its positive part is fully satisfied and none of its default-negated literals is contradicted.

Definition 4 (Applicability of a Heuristic Directive) *A ground heuristic directive h is* applicable *w.r.t. a partial assignment A iff:* $\mathsf{cond}(h)$ *is satisfied, and* $\mathsf{truth}_A(\mathsf{atm}(\mathsf{head}(h))) \in \{\mathbf{U}, \mathbf{M}\}$.

Intuitively, a heuristic directive is applicable iff its condition is satisfied and the atom in its head is assigned neither $\mathbf{T}$ nor $\mathbf{F}$. If the atom in the head is assigned $\mathbf{M}$, the heuristic directive may still be applicable, because any atom with the non-final truth value $\mathbf{M}$ must be either $\mathbf{T}$ or $\mathbf{F}$ in any answer set.

Definitions 2 to 4 reveal the distinguishing features of the newly proposed semantics: In our approach, heuristic signs composed of truth values $\mathbf{T}$, $\mathbf{M}$, and $\mathbf{F}$ can be used in heuristic conditions to reason about atoms that are already assigned specific truth values in a partial assignment. Furthermore, default negation can be used to reason about atoms that are assigned *or still unassigned*. Our semantics truly means default negation in the current partial assignment, while the one implemented by CLINGO amounts to strong negation in the current search state. This difference is crucial since reasoning about incomplete information is essential in many cases. An example is a heuristic for a configuration problem that only applies to components not yet placed.

[5] In the head, we only support truth values $\mathbf{T}$ and $\mathbf{F}$ because, from a user's point of view, it does not make sense to assign $\mathbf{M}$ to an atom heuristically.

[6] Note that the function truth maps to only one truth value even though $\mathbf{T}a \in A$ implies $\mathbf{M}a \in A$, so $\mathsf{truth}_A(a) = \mathbf{M}$ iff $\mathbf{M}a \in A$ and $\mathbf{T}a \notin A$.

What remains to be defined is the semantics of weight and level. Given a set of applicable heuristic directives, one directive with the highest weight will be chosen from the highest level. Suppose there are several with the same maximum priority (i.e., weight and level). In that case, the solver can use domain-independent heuristics like VSIDS [26] as a fallback to break the tie.

Definition 5 (Heuristics Eligible for Immediate Choice) *Given a set H of applicable ground heuristic directives, the* subset eligible for immediate choice *is defined as* $\mathsf{maxpriority}(H)$ *in two steps:*

$$\begin{aligned}
\mathsf{maxlevel}(H) :=\ & \{h \mid h \in H \text{ and } \mathsf{level}(h) = \max_{h \in H} \mathsf{level}(h)\} \\
\mathsf{maxpriority}(H) :=\ & \{h \mid h \in \mathsf{maxlevel}(H) \text{ and} \\
& \quad \mathsf{weight}(h) = \max_{h \in \mathsf{maxlevel}(H)} \mathsf{weight}(h)\}
\end{aligned}$$

After choosing a heuristic using $\mathsf{maxpriority}$, a solver makes a decision on the directive's head atom. Note that heuristics only choose between atoms derivable by currently applicable rules. Other solving procedures, e.g., deterministic propagation, are unaffected by processing heuristics.

Example 3 *Consider the following program.*

$$\begin{array}{lr}
\{\ \mathrm{a}(2)\ ;\ \mathrm{a}(4)\ ;\ \mathrm{a}(6)\ ;\ \mathrm{a}(8)\ ;\ \mathrm{a}(5)\ \} \leftarrow . & \\
\leftarrow \#\mathtt{sum}\ \{\ X : \mathrm{a}(X)\ \} = S,\ \ S \backslash 2 \neq 0. & \\
\#\mathtt{heuristic}\ \ \mathrm{a}(5).\ [1] & \langle 2 \rangle \\
\#\mathtt{heuristic}\ \ \mathrm{a}(4)\ :\ \mathrm{not}\ \mathrm{a}(5).\ [2] & \langle 3 \rangle \\
\#\mathtt{heuristic}\ \ \mathbf{F}\mathrm{a}(5)\ :\ \mathrm{a}(4).\ [2] & \langle 4 \rangle \\
\#\mathtt{heuristic}\ \ \mathrm{a}(6)\ :\ \mathbf{F}\mathrm{a}(5), \mathbf{T}\mathrm{a}(4).\ [2] & \langle 5 \rangle
\end{array}$$

Intuitively, directive ⟨2⟩ unconditionally prefers to make a(5) *true with weight 1. All other directives have a higher weight, 2, but they become applicable at different times. Directive ⟨3⟩ prefers to make* a(4) *true if* a(5) *is neither true nor must-be-true, directive ⟨4⟩ prefers to make* a(5) *false if* a(4) *is true or must-be-true, and ⟨5⟩ prefers to make* a(6) *true if* a(5) *is false and* a(4) *is true.*

Let $A_0 = \emptyset$ be the empty partial assignment before any decision has been made. W.r.t. A_0, ⟨2⟩ is applicable because its condition is empty and its head is still unassigned. Directive ⟨3⟩ is also applicable because a(5) *is still unassigned. Directives ⟨4⟩ and ⟨5⟩ are not applicable w.r.t. A_0. Directive ⟨3⟩ is chosen because it has the highest priority among applicable directives. Thus,* a(4) *is assigned* $\mathbf{T}$, *updating our assignment to* $A_1 = \{\mathbf{M}\mathrm{a}(4), \mathbf{T}\mathrm{a}(4)\}$. *This makes ⟨4⟩ applicable,* a(5) *is assigned* $\mathbf{F}$ *and our assignment is* $A_2 = \{\mathbf{M}\mathrm{a}(4), \mathbf{T}\mathrm{a}(4), \mathbf{F}\mathrm{a}(5)\}$. *Note that the condition of ⟨3⟩ was still satisfied at this point, but it was not applicable because its head was already assigned. Now, ⟨2⟩ is also not applicable anymore, and the only directive that remains is ⟨5⟩. Since ⟨5⟩ is applicable,* a(6) *is made true and added to the assignment. Next, the atoms that remained unassigned are guessed by the default heuristics until an answer set is found.*

5 APPLICATIONS AND EXPERIMENTS

In this section, we present an application of our approach on two configuration problems. Experimental results are also included, using an implementation of our approach in the lazy-grounding ASP system ALPHA. Instance sets include instances that were challenging to ground and solve.

5.1 Experimental Setup

Encodings (including heuristics) and instances, and the ALPHA binaries used for our experiments, are available on our website.[7] Details on the sources of the encodings are mentioned in the sections describing the domains. Optimisation statements were not used since ALPHA does not support them yet. However, heuristic directives can be written in a way that optimal or near-optimal solutions are preferably found.

Problem instances were selected by first defining an instance-generating algorithm and then exploring instance sizes to find a set in which all systems could solve some instances under consideration within a time limit of 10 minutes, and some instances could be solved by none (or very few) of these systems.

The ASP systems used for the experiments were ALPHA[8], CLINGO[9] [15] version 5.4.0 and DLV2[10] [1] version 2.1.0.

ALPHA is used in two configurations, grounding constraints either strictly or permissively [30]. Traditionally in lazy grounding, rules are grounded when their positive body is fully satisfied (*strict* lazy grounding). *Permissive* grounding, on the other hand, enables rules to be grounded if their positive body is not fully satisfied, as long as all variables can be bound by positive body literals that are already satisfied. For example, consider the following non-ground constraint:

$$\leftarrow \mathrm{a}(X, Y), \mathrm{b}(X).$$

Under the partial assignment $A = \{\mathbf{M}\mathrm{a}(1,2), \mathbf{T}\mathrm{a}(1,2)\}$, the ground constraint $\leftarrow \mathrm{a}(1,2), \mathrm{b}(1).$ will only be produced if permissive grounding of constraints is enabled.

All solvers were configured to search for the first answer set of each problem instance. Finding one or only a few solutions is often sufficient in industrial use cases since solving large instances can be challenging [10]. Therefore, the domain-specific heuristics used in the experiments are designed to help the solver find one answer set that is "good enough", even though it may not be optimal.

5.2 The House Reconfiguration Problem (HRP)

The House Reconfiguration Problem [12] is an abstracted version of industrial (re)configuration problems, e.g., rack configuration.

Formally, HRP is defined as a modification of the House Configuration Problem (HCP).

Definition 6 (HCP) *The input for the* House Configuration Problem (HCP) *is given by four sets of constants P, T, C, and R representing persons, things, cabinets, and rooms, respectively, and an ownership relation $PT \subseteq P \times T$ between persons and things.*

The task is to find an assignment of things to cabinets $TC \subseteq T \times C$ and cabinets to rooms $CR \subseteq C \times R$, such that: (1) each thing is stored in a cabinet; (2) a cabinet contains at most five things; (3) every cabinet is placed in a room; (4) a room contains at most four cabinets; and (5) a room may only contain cabinets storing things of one person.

Definition 7 (HRP) *The input for the* House Reconfiguration Problem (HRP) *is given by an HCP instance $H = \langle P, T, C, R, PT \rangle$, a legacy configuration $\langle TC', CR' \rangle$, and a set of things $T' \subseteq T$ that are defined as "long" (all other things are "short").*

The task is then to find an assignment of things to cabinets $TC \subseteq T \times C$ and cabinets to rooms $CR \subseteq C \times R$, that satisfies all requirements of HCP as well as the following ones: (1) a cabinet is either small or high; (2) a long thing can only be put into a high cabinet; (3) a small cabinet occupies 1 and a high cabinet 2 of 4 slots available in a room; (4) all legacy cabinets are small.

The sample HRP instance shown in Fig. 1 comprises two cabinets, two rooms, five things that belong to person p_1, and one thing that belongs to person p_2. A legacy configuration is empty, and all things are small. In a solution, the first person's things are placed in cabinet c_1 in the first room, and the thing of the second person is the cabinet c_2 in the second room. For this sample instance, a solution of HRP corresponds to a solution of HCP.

In the problem encoding [12], two main choice rules are responsible for guessing the assignment of things to cabinets and the assignment of cabinets to rooms:

$$\{\ \mathrm{cabinetTOthing}(C, T)\ \} \leftarrow \mathrm{cabinetDomain}(C), \mathrm{thing}(T).$$
$$\{\ \mathrm{roomTOcabinet}(R, C)\ \} \leftarrow \mathrm{roomDomain}(R), \mathrm{cabinet}(C)$$

Instances consist of facts over the following predicates: $\mathrm{cabinetDomain}/1$ defines potential cabinets and $\mathrm{roomDomain}/1$ defines potential rooms; $\mathrm{thingLong}/1$ defines which things are long; and $\mathrm{legacy}/1$ defines all the other data in the legacy configuration, e.g., $\mathrm{legacy}(\mathrm{personTOthing}(\mathrm{p1}, \mathrm{t1}))$ defines that person p1 owns thing t1, and $\mathrm{legacy}(\mathrm{roomTOcabinet}(\mathrm{r1}, \mathrm{c1}))$ specifies one tuple in the legacy assignment of cabinets to rooms.

The domain-specific heuristics for HRP implemented in our novel approach works by (1) first trying to re-use the legacy configuration; (2) then filling cabinets with things; (3) then filling rooms with cabinets; (4) and finally closing remaining choices. Long things are always assigned before short things.

By "closing remaining choices" we mean assigning $\mathbf{F}$ to choice points not yet assigned by the heuristics. The purpose of this is to avoid the default heuristics (e.g., VSIDS) from causing conflicts by choosing the wrong truth values.

We now present some selected heuristic directives. The directives use some intermediate predicates whose meaning should become evident from their names. The full encoding is available online. The following heuristics re-use the legacy assignment of cabinets to things and of rooms to cabinets (1):

$$\texttt{\#heuristic}\ \mathrm{reuse}(\mathrm{cabinetTOthing}(C, T)) :$$
$$\mathrm{legacy}(\mathrm{cabinetTOthing}(C, T)), \mathrm{thingLong}(T).\quad [4@4]$$

$$\texttt{\#heuristic}\ \mathrm{reuse}(\mathrm{cabinetTOthing}(C, T)) :$$
$$\mathrm{legacy}(\mathrm{cabinetTOthing}(C, T)), \text{not } \mathrm{thingLong}(T).\quad [3@4]$$

$$\texttt{\#heuristic}\ \mathrm{reuse}(\mathrm{roomTOcabinet}(R, C)) :$$
$$\mathrm{legacy}(\mathrm{roomTOcabinet}(R, C)).\quad [2@4]$$

The following heuristic assigns things to cabinets, preferring long over short things (2):

$$\begin{aligned}&\texttt{\#heuristic}\ \mathrm{cabinetTOthing}(C, T) :\\ &\quad \mathrm{cabinetDomain}(C), \text{not } \mathrm{fullCabinet}(C),\\ &\quad \text{not } \mathbf{T}\ \mathrm{assignedThing}(T), \mathrm{personTOthing}(P, T),\\ &\quad \text{not } \mathrm{otherPersonTOcabinet}(P, C),\\ &\quad \mathrm{maxCabinet}(MC), \mathrm{thingLong}(T).\ [(MC - C)@3]\end{aligned}$$

$$\begin{aligned}&\texttt{\#heuristic}\ \mathrm{cabinetTOthing}(C, T) :\\ &\quad \mathrm{cabinetDomain}(C), \text{not } \mathrm{fullCabinet}(C),\\ &\quad \text{not } \mathbf{T}\ \mathrm{assignedThing}(T), \mathrm{personTOthing}(P, T)\\ &\quad \text{not } \mathrm{otherPersonTOcabinet}(P, C),\\ &\quad \mathrm{maxCabinet}(MC), \text{not } \mathrm{thingLong}(T).\ [(MC - C)@2]\end{aligned}$$

[7] `https://ainf.aau.at/dynacon`
[8] `https://github.com/alpha-asp/Alpha`
[9] `https://potassco.org/clingo/`
[10] `https://dlv.demacs.unical.it/`

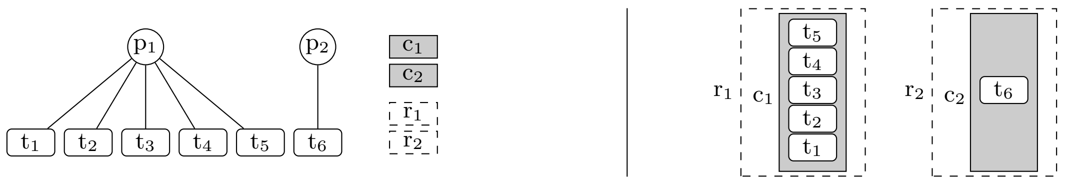
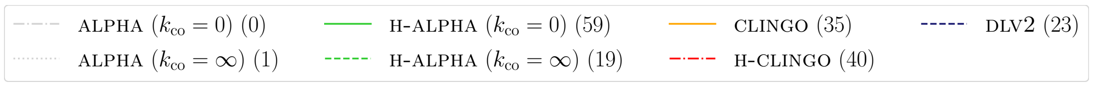
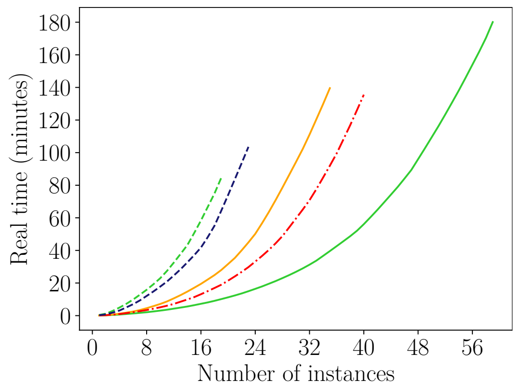


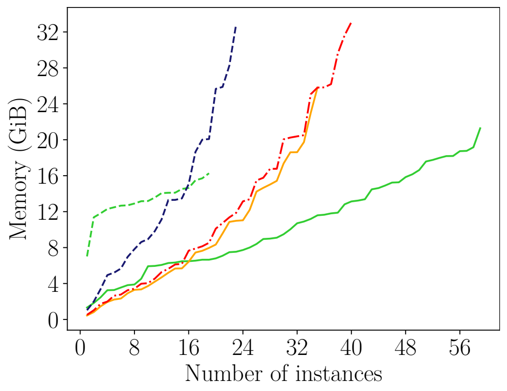
Figure 1. Sample HRP instance (left) and one of its solutions (right)



(a) Solver configurations, with numbers of solved instances



(b) Accumulated time consumption



(c) Memory consumption

Figure 2. Time and memory consumption for solving each HRP instance

The following heuristic assigns cabinets to rooms (3):

`#heuristic` $\mathrm{roomTOcabinet}(R, C)$:
$\mathrm{roomDomain}(R), \text{not } \mathrm{fullRoom}(R),$
$\mathrm{cabinet}(C), \text{not } \mathbf{T} \ \mathrm{assignedCabinet}(C),$
$\mathrm{personTOcabinet}(P, C), \text{not } \mathrm{otherPersonTOroom}(P, R),$
$\mathrm{maxRoom}(MR). \ [(MR - R)@1]$

Finally, we close choice points that are still unassigned (4):

`#heuristic` $\mathbf{F} \ \mathrm{cabinetTOthing}(C, T)$:
$\text{not } \mathrm{cabinetTOthing}(C, T), \mathrm{cabinetDomain}(C), \mathrm{thing}(T).$

`#heuristic` $\mathbf{F} \ \mathrm{roomTOcabinet}(R, C)$:
$\text{not } \mathrm{roomTOcabinet}(R, C), \mathrm{roomDomain}(R), \mathrm{cabinet}(C).$

The heuristics we created for ALPHA cannot be used with CLINGO due to the usage of **T** and default negation. An alternative encoding containing faithfully adapted heuristic directives for CLINGO has also been created.

Figure 2 shows performance data for experiments with HRP. Cactus plots were created in the usual way. In Fig. 2b, the x-axis gives the number of instances solved within real (i.e., wall-clock) time, given on the y-axis. Time is accumulated over all solved instances. Memory consumption is given on the y-axis of Fig. 2c, where data points are sorted by y-values, which are not accumulated. Figure 2a contains a legend with all solver configurations. The number of instances solved by each system is shown next to its name (in parentheses).

One curve was drawn for each solver configuration: ALPHA without domain-specific heuristics, with strict ($k_{co} = 0$) and permissive ($k_{co} = \infty$) grounding of constraints; ALPHA with domain-specific heuristics (H-ALPHA), with strict and permissive grounding of constraints; DLV2; CLINGO with (H-CLINGO) and without domain-specific heuristics.

Substantial differences can be observed. The curves for H-ALPHA ($k_{co} = 0$) reach farthest to the right, meaning that ALPHA with domain-specific heuristics solved the highest number of instances (59 out of 94) when grounding constraints strictly. Surprisingly, with permissive grounding of constraints, ALPHA with domain-specific heuristics exhibited relatively low time and space performance.

No curves are visible at all for ALPHA without domain-specific heuristics because, in this configuration, the system could solve at most one instance only. The other solvers' performance was somewhere in between the ALPHA configurations at both ends of the spectrum. Notably, H-CLINGO with domain-specific heuristics solved more instances in less time compared to CLINGO without domain-specific heuristics, but consumed slightly more memory. The largest instance solved by H-ALPHA contained 600 things, which is over 30% more than the size of the largest instance solved by H-CLINGO (455). Recall that the time limit for solving each instance was 10 minutes.

5.3 The Partner Units Problem (PUP)

Like HRP, the Partner Units Problem (PUP) [31, 34] is an abstracted version of industrial (re)configuration problems.

Definition 8 (PUP) *The input to the* Partner Units Problem (PUP) *is given by a set of units U and a bipartite graph $G = (S, Z, E)$, where S is a set of sensors, Z is a set of zones, and E is a relation between S and Z.*

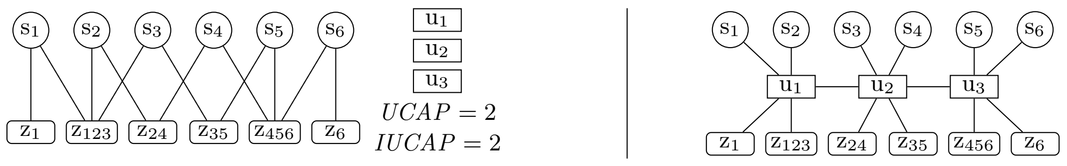
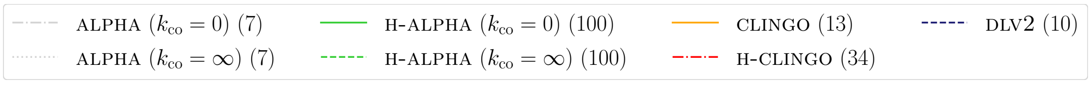
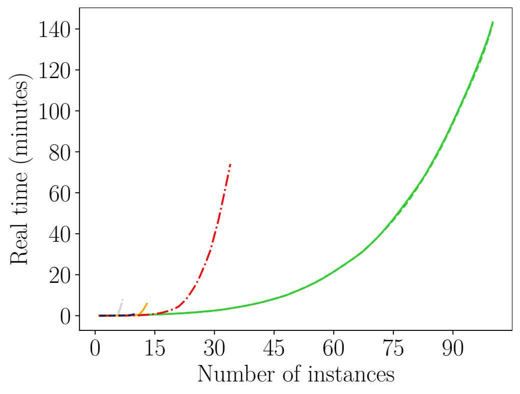


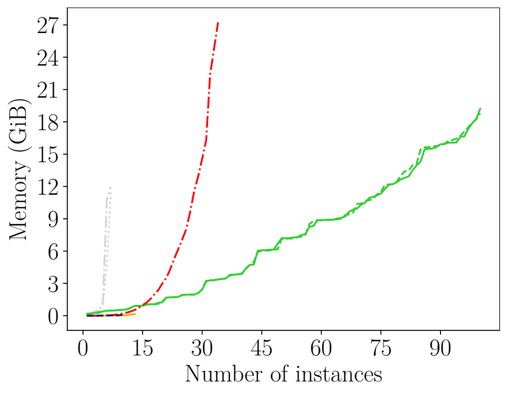
Figure 3. Sample PUP instance (left) and one of its solutions (right)



(a) Solver configurations, with numbers of solved instances



(b) Accumulated time consumption



(c) Memory consumption

Figure 4. Time and memory consumption for solving each PUP instance

The task is to find a partition of vertices $v \in S \cup Z$ into bags $u_i \in U$ such that for each bag the following requirements hold: (1) the bag contains at most $UCAP$ vertices from S and at most $UCAP$ vertices from Z; and (2) the bag has at most $IUCAP$ adjacent bags, where the bags u_1 and u_2 are adjacent whenever $v_i \in u_1$ and $v_j \in u_2$ for some $(v_i, v_j) \in E$.

Figure 3 shows an example of a PUP instance. The bipartite graph comprises six sensors and six zones. Each of the three units can be adjacent to at most two other units, and each unit can contain at most two sensors and two zones. An assignment of sensors and zones to units that satisfies all PUP requirements is also presented in Fig. 3.

PUP instances consist of atoms over the predicates $\mathrm{comUnit}/1$ (specifying units U) and $\mathrm{zone2sensor}/2$ (specifying the zone-to-sensor relation E).[11]

Our encoding for PUP is based on encodings from the ASP Competitions [5,6]. The following rules constitute the main guessing part of the encoding:

$$\mathrm{elem}(\mathrm{z}, Z) \leftarrow \mathrm{zone2sensor}(Z, D).$$
$$\mathrm{elem}(\mathrm{s}, D) \leftarrow \mathrm{zone2sensor}(Z, D).$$
$$\{ \mathrm{assign}(U, T, X) \} \leftarrow \mathrm{elem}(T, X), \mathrm{comUnit}(U).$$

QuickPup is a heuristic for PUP that successfully solves many hard problem instances [33]. Our approach supports implementing large parts of the originally procedural algorithm for QuickPup. Our encoding uses rules to derive a topological order of the zones and sensors (cf. [31,32]). Heuristic directives subsequently use this topological order.

[11] In the instances used for our experiments, both $UCAP$ and $IUCAP$ are fixed at the value 2.

First, a start zone is determined and denoted by $\mathrm{startZone}/1$. In our encoding, the start zone is always the first one and instances are designed so that solutions can easily be found when starting with this zone. QuickPup should actually try to use each zone as the start zone one after the other and abort search after a certain amount of time has passed. This part of the algorithm cannot currently be represented in our framework.

QuickPup assigns zones and sensors to units in a breadth-first-order, called "topological order" because the graph is traversed level by level. First, the start zone is assigned, then the sensors connected to the start zone, then the zones connected to those sensors and so on. A helper predicate $\mathrm{layer}/3$ is introduced to compute the topological order. In an atom $\mathrm{layer}(T, X, L)$, T denotes the type of element ("s" for sensor and "z" for zone), X is the element's identifier, and L is its layer in the computed breadth-first order.

A realisation of QuickPup in our framework requires several heuristic directives. These directives use the level term in the annotation to process the zones and sensors according to the topological order. The weight term in the annotation is used to assign the units in the right order. The directives use some intermediate predicates whose meaning should become evident from their names. The full encoding is available online.

QuickPup first tries to assign an element to the first unit:

$$\begin{array}{l} \#\mathtt{heuristic}\ \mathrm{assign}(U, T, X) : \ \mathrm{comUnit}(U), \mathrm{elem}(T, X), \\ \quad \mathrm{maxLayer}(ML), \mathrm{layer}(T, X, L), \\ \quad \mathrm{not\ full}(U, T), \mathrm{not\ assigned}(T, X), \mathrm{not}\ \mathbf{T}\ \mathrm{used}(U), \\ \quad \mathrm{nUnits}(NU), U = 1.\ [NU@(ML - L)] \end{array}$$

If one unit cannot be assigned, QuickPup tries preceding units in decreasing order next:

$\texttt{\#heuristic}\ \text{assign}(U, T, X) :\ \text{comUnit}(U), \text{elem}(T, X),$
$\quad \text{maxLayer}(ML), \text{layer}(T, X, L),$
$\quad \text{not full}(U, T), \text{not assigned}(T, X),$
$\quad \mathbf{T}\ \text{used}(U).\ [U@(ML - L)]$

A fresh unit is only touched if all preceding units have been tried:
$\texttt{\#heuristic}\ \text{assign}(U, T, X) :\ \text{comUnit}(U), \text{elem}(T, X),$
$\quad \text{maxLayer}(ML), \text{layer}(T, X, L),$
$\quad \text{not full}(U, T), \text{not assigned}(T, X),$
$\quad \text{not}\ \mathbf{T}\ \text{used}(U), \text{comUnit}(U - 1), \mathbf{T}\ \text{used}(U - 1),$
$\quad \mathbf{F}\ \text{assign}(U - 1, T, X).\ [U@(ML - L)]$

Note the condition $\mathbf{F}\ \text{assign}(\mathrm{U} - 1, \mathrm{T}, \mathrm{X})$ in the last heuristic directive. Due to this condition, the heuristic is only applicable if the same element could not be assigned to the preceding unit $U - 1$. This situation may be caused by backtracking or by the following heuristic avoiding assignments to units that are already full:
$\texttt{\#heuristic}\ \mathbf{F}\ \text{assign}(U, T, X) :\ \text{comUnit}(U), \text{elem}(T, X),$
$\quad \text{maxLayer}(ML), \text{full}(U, T),$
$\quad \text{not assign}(U, T, X).\ [1@ML]$

Choice points not assigned by any of these heuristics are finally assigned false by a dedicated heuristic directive, similarly as shown for HRP in Section 5.2.

The heuristics we created for ALPHA cannot be used with CLINGO due to the usage of **T**, **F**, and default negation. An alternative encoding containing heuristic directives for CLINGO has been created that is similar to the QuickPup*-like heuristics created for ALPHA.

Cactus plots for PUP (Fig. 4) were generated in the same way as for HRP (cf. Section 5.2).

Again, ALPHA with domain-specific heuristics solved the highest number of instances (all 100). The curves for H-ALPHA ($k_{\text{co}} = 0$) and H-ALPHA ($k_{\text{co}} = \infty$) are almost indistinguishable, meaning that it did not make a difference in the PUP domain whether constraints were grounded strictly or permissively.

At the other extreme, ALPHA without domain-specific heuristic could solve only 7 of the 100 instances.

The systems DLV2, CLINGO, and H-CLINGO performed somewhere between those extremes. H-CLINGO with domain-specific heuristics solved many more instances than CLINGO without domain-specific heuristics.

The largest instance in our instance set contained 300 units. H-ALPHA was able to solve all these instances. In contrast, the size of the largest instance that could be solved by any other system, using the given encoding, was only 105. Recall that the time limit for solving each instance was 10 minutes. For 11 instances, H-CLINGO returned an error ("Id out of range").

5.4 Discussion

Our results show that we have extended the application area of ASP. By combining our novel approach to domain-specific heuristics with lazy-grounding answer set solving, we could solve large-scale problem instances that are out of reach for conventional ASP systems. This finding supports our initial hypothesis that both lazy grounding and domain-specific heuristics are crucial for solving large-scale industrial problems.

Our approach extends an earlier extension of ASP's input language by a declarative framework for domain-specific heuristics [17]. Our advancement consists of novel syntax and semantics for heuristic directives that make it possible to reason about the current partial assignment, facilitating heuristics based on what has or has *not yet* been decided by the solver. Although the earlier approach has worked very well on planning problems, it seems that more flexibility in the definition of heuristics, supported by the novel features of our approach, is necessary to represent heuristics for other kinds of problems.

Our results undeniably show that domain-specific heuristics improve solving performance for the domains under consideration. This is not only true for ALPHA but also for CLINGO. However, domain-specific heuristics usually increase CLINGO's memory consumption, thus exacerbating the grounding bottleneck from which ground-and-solve systems such as CLINGO are suffering. Domain-specific heuristics for DLV2 were out of scope because DLV2 does not support the declarative specification of heuristics.

Solution quality is another aspect to keep in mind. Since domain-specific heuristics lead the solver towards "better" solutions, the quality of answer sets computed by H-ALPHA or H-CLINGO is higher than the quality of answer sets computed by ALPHA, CLINGO, or DLV2.

However, we do not claim that heuristics based on partial assignments are always beneficial. Our findings cannot reject the possibility that H-CLINGO might outperform H-ALPHA when other encodings or other heuristics are used since there might be encoding optimisations that we have not thought of. Still, we are confident that our approach's novel features make the specification of practical heuristics more intuitive and effortless.

Results for HRP (Fig. 2) indicate that permissive grounding (cf. [30]), i.e., providing the solver with more nogoods representing ground constraints than necessary, can be counterproductive when domain-specific heuristics are used. We conjecture the reason for this to be that suitable domain-specific heuristics can assist the solver even better than additional constraints while avoiding the overhead of additional nogoods (in terms of space consumption and propagation efforts). This assumption is supported by the considerable increase in ALPHA's memory consumption when grounding constraints permissively in this domain.

To sum up, domain-specific heuristics implemented in our novel framework, combined with strict lazy grounding by ALPHA, outperformed all other tested systems when applied to large instances of the House Reconfiguration Problem and the Partner Units Problem. Applications to other domains should be easy to put into practice and belong to future work.[12]

6 CONCLUSION

We have proposed novel syntax and semantics for declarative domain-specific heuristics in ASP that can depend non-monotonically on the partial assignment maintained during solving. Furthermore, we have presented experimental results obtained with an implementation of our approach within the lazy-grounding solver ALPHA.

Our semantics has proven beneficial for several practical application domains, advancing CLINGO's previous approach [17]. In experiments, our implementation exhibited convincing time and memory consumption behaviour. Thus, we extended the application area of ASP by solving large configuration problem instances that conventional ASP systems could not solve.

Our approach's suitability to implement heuristics for other practical (configuration) problems should be assessed by the community.

[12] One other domain, A* search, is investigated in our journal paper [28].

Some real-world domain-specific heuristics will require extensions of our approach, such as by supporting randomness and restarts.

Thinking more broadly, the question of how to generate domain-specific heuristics automatically is of great importance since, currently, such heuristics have to be invented by humans familiar with the domain (and partly also with solving technology).

ACKNOWLEDGEMENTS

This work has been conducted in the scope of the research project *DynaCon (FFG-PNr.: 861263)*, which was funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between 2017 and 2020.[13] This research was also supported by the Academy of Finland, project 251170, and by EU ECSEL Joint Undertaking under grant agreement no. 737459 (project Productive4.0).

We are thankful to Peter Schüller for his contributions to our earlier conference paper on this topic, and Stephen Cummings for language editing.

REFERENCES

[1] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari, 'The ASP system DLV2', in *LPNMR*, volume 10377 of *LNCS*, pp. 215–221. Springer, (2017).

[2] Chitta Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2003.

[3] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub, 'ASP-Core-2 input language format', *TPLP*, **20**(2), 294–309, (2020).

[4] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari, 'I-DLV: the new intelligent grounder of DLV', *Intelligenza Artificiale*, **11**(1), 5–20, (2017).

[5] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca, 'Design and results of the fifth answer set programming competition', *Artif. Intell.*, **231**, 151–181, (2016).

[6] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca, 'The third open answer set programming competition', *TPLP*, **14**(1), 117–135, (2014).

[7] Carmine Dodaro, Philip Gasteiger, Nicola Leone, Benjamin Musitsch, Francesco Ricca, and Konstantin Schekotihin, 'Combining answer set programming and domain heuristics for solving hard industrial problems (application paper)', *TPLP*, **16**(5-6), 653–669, (2016).

[8] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran, 'Complexity results for answer set programming with bounded predicate arities and implications', *Ann. Math. Artif. Intell.*, **51**(2-4), 123–165, (2007).

[9] Esra Erdem, Michael Gelfond, and Nicola Leone, 'Applications of answer set programming', *AI Magazine*, **37**(3), 53–68, (2016).

[10] Andreas A. Falkner, Gerhard Friedrich, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, 'Twenty-five years of successful application of constraint technologies at Siemens', *AI Magazine*, **37**(4), 67–80, (2016).

[11] Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan, 'Industrial applications of answer set programming', *KI*, **32**(2-3), 165–176, (2018).

[12] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, '(Re)configuration using answer set programming', in *IJCAI 2011 Workshop on Configuration*, volume 755 of *CEUR Workshop Proceedings*, pp. 17–24, (2011).

[13] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, and Philipp Wanko, *Potassco guide version 2.2.0*, 2019. Retrieved from `https://github.com/potassco/guide/releases/tag/v2.2.0`.

[14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub, *Answer Set Solving in Practice*, Morgan and Claypool Publishers, 2012.

[15] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub, 'Clingo = ASP + control: Preliminary report', *CoRR*, **abs/1405.3694**, (2014).

[16] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub, 'Advances in *gringo* series 3', in *LPNMR*, volume 6645 of *LNCS*, pp. 345–351. Springer, (2011).

[17] Martin Gebser, Benjamin Kaufmann, Javier Romero, Ramón Otero, Torsten Schaub, and Philipp Wanko, 'Domain-specific heuristics in answer set programming', in *AAAI*. AAAI Press, (2013).

[18] Martin Gebser, Marco Maratea, and Francesco Ricca, 'The seventh answer set programming competition: Design and results', *TPLP*, **20**(2), 176–204, (2020).

[19] Martin Gebser, Anna Ryabokon, and Gottfried Schenner, 'Combining heuristics for configuration problems using answer set programming', in *LPNMR*, volume 9345 of *LNCS*, pp. 384–397. Springer, (2015).

[20] Michael Gelfond and Yulia Kahl, *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*, Cambridge University Press, 2014.

[21] Lothar Hotz, Alexander Felfernig, Markus Stumptner, Anna Ryabokon, Claire Bagley, and Katharina Wolter, 'Chapter 6 - Configuration Knowledge Representation and Reasoning', in *Knowledge-Based Configuration*, 41–72, Morgan Kaufmann, (2014).

[22] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello, 'The DLV system for knowledge representation and reasoning', *ACM Trans. Comput. Log.*, **7**(3), 499–562, (2006).

[23] Nicola Leone and Francesco Ricca, 'Answer set programming: A tour from the basics to advanced development tools and industrial applications', in *Reasoning Web*, eds., Wolfgang Faber and Adrian Paschke, volume 9203 of *LNCS*, pp. 308–326. Springer, (2015).

[24] Lorenz Leutgeb and Antonius Weinzierl, 'Techniques for efficient lazy-grounding ASP solving', in *DECLARE*, volume 10997 of *LNCS*, pp. 132–148. Springer, (2017).

[25] Vladimir Lifschitz, *Answer Set Programming*, Springer, 2019.

[26] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, 'Chaff: Engineering an efficient SAT solver', in *38th Design Automation Conference (DAC)*, pp. 530–535. ACM, (2001).

[27] Gottfried Schenner and Richard Taupe, 'Techniques for solving large-scale product configuration problems with ASP', in *Proceedings of the 19th International Configuration Workshop*, eds., Linda L. Zhang and Albert Haag, pp. 12–19, La Défense, France, (2017).

[28] Richard Taupe, Gerhard Friedrich, Konstantin Schekotihin, and Antonius Weinzierl, 'Domain-specific heuristics in answer set programming: A declarative non-monotonic approach'. Under review.

[29] Richard Taupe, Konstantin Schekotihin, Peter Schüller, Antonius Weinzierl, and Gerhard Friedrich, 'Exploiting partial knowledge in declarative domain-specific heuristics for ASP', in *ICLP Technical Communications*, volume 306 of *EPTCS*, pp. 22–35, (2019).

[30] Richard Taupe, Antonius Weinzierl, and Gerhard Friedrich, 'Degrees of laziness in grounding – effects of lazy-grounding strategies on ASP solving', in *LPNMR*, volume 11481 of *LNCS*, pp. 298–311, (2019).

[31] Erich C Teppan, 'Solving the partner units configuration problem with heuristic constraint answer set programming', in *Configuration Workshop*, pp. 61–68, (2016).

[32] Erich Christian Teppan and Gerhard Friedrich, 'Heuristic constraint answer set programming', in *ECAI*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pp. 1692–1693. IOS Press, (2016).

[33] Erich Christian Teppan, Gerhard Friedrich, and Andreas A. Falkner, 'QuickPup: A heuristic backtracking algorithm for the partner units configuration problem', in *IAAI*. AAAI, (2012).

[34] Erich Christian Teppan, Gerhard Friedrich, and Georg Gottlob, 'Tractability frontiers of the partner units configuration problem', *J. Comput. Syst. Sci.*, **82**(5), 739–755, (2016).

[35] Antonius Weinzierl, 'Blending lazy-grounding and CDNL search for answer-set solving', in *LPNMR*, volume 10377 of *LNCS*, pp. 191–204. Springer, (2017).

[13] See `https://iktderzukunft.at/en/` for more information.