TPLP: Page 1–16.
doi:10.1017/S147106842510029X

*Smart Expansion Techniques for ASP-Based Interactive Configuration**

LUCIA BALÁŽOVÁ and RICHARD COMPLOI-TAUPE
Siemens AG Österreich, Vienna, Austria
(*e-mail:* richard.taupe@siemens.com)

SUSANA HAHN
University of Potsdam, Potsdam, Germany
Potassco Solutions, Werder (Havel), Germany
(*e-mail:* hahnmartinlu@uni-potsdam.de)

NICOLAS RÜHLING
University of Potsdam, Potsdam, Germany
UP Transfer, Potsdam, Germany
(*e-mail:* nruehling@uni-potsdam.de)

GOTTFRIED SCHENNER
SIEMENS AG Österreich, Austria
(*e-mail:* gottfried.schenner@siemens.com)

submitted 20 July 2025; revised 20 July 2025; accepted 27 July 2025

Abstract

Product configuration is a successful application of answer set programming (ASP). However, challenges are still open for interactive systems to effectively guide users through the configuration process. The aim of our work is to provide an ASP-based solver for interactive configuration that can deal with large-scale industrial configuration problems and that supports intuitive user interfaces (UIs) via an application programming interface (API). In this paper, we focus on improving the performance of automatically completing a partial configuration. Our main contribution enhances the classical incremental approach for multi-shot solving by four different smart expansion functions. The core idea is to determine and add specific objects or associations to the partial configuration by exploiting cautious and brave consequences before checking for the existence of a complete configuration with the current objects in each iteration. This approach limits the number of costly unsatisfiability checks and reduces the search space, thereby improving solving performance. In addition, we present a UI that uses our API and is implemented in ASP.

KEYWORDS: answer set programming, product configuration, interactivity

* This research was partially supported by the Austrian Research Promotion Agency (FFG) under the "AI for Green" program and by the German Federal Ministry for Economic Affairs and Energy (BMWE) through the ZIM project grant KK5291307GR4.

1 Introduction

Configuration (Felfernig *et al.* 2014) has been one of the first successful applications (Gebser *et al.* 2011b; Felfernig *et al.* 2017; Gençay *et al.* 2019) of answer set programming (ASP; Gelfond and Kahl (2014); Lifschitz (2019)). Nonetheless, there are open challenges for ASP-based configurators, one of them being interactive configuration.

Industrial configuration problems in large infrastructure projects may contain thousands of components and hundreds of component types and are typically solved in a step-wise manner by combining interactive actions with automatic solving of subproblems (Falkner *et al.* 2020). When using a grounding-based formalism like ASP, a grounding bottleneck (Eiter *et al.* 2007) arises due to the large number of components. Furthermore, the necessary domain size is not known beforehand and can vary significantly. Therefore, we require a way to dynamically introduce new components during the configuration process.

In our previous work, we developed an application programming interface (API) to satisfy basic requirements for interactive configuration (Falkner *et al.* 2020) based on OOASP (Falkner *et al.* 2015) and using various features of *clingo*[1] (Gebser *et al.* 2019). In this paper, we report on our recent advancements in developing an ASP-based domain-independent interactive configuration platform. The new achievements, compared to previously published reports (Comploi-Taupe *et al.* 2022, 2023), consist of dramatically improved solving performance by using novel so-called *smart expansion* techniques and the creation of a user interface (UI) prototype based on *clinguin* (Beiser *et al.* 2024).

The general idea of the smart expansion techniques is to infer as much knowledge as possible from the current configuration state and to use this knowledge to reduce the search space of the ASP solver and to limit costly unsatisfiability checks. This results from the insight that in our earlier implementation, objects were added incrementally to the configuration without any class restrictions or associations, which in turn led to a combinatorial explosion of the search space. For this purpose, we introduced four new functions: *ObjectNeeded*, *GlobalUpperBoundGap*, *GlobalLowerBoundGap*, and *AssociationPossible*. While the first three ones use intersections of possible solutions to determine objects that are definitely needed in the configuration, the latter function provides possible associations between two objects with the intention of quickly estimating the minimal size of a complete configuration. We demonstrate the working of the functions using our racks example as known from earlier work.

The remainder of the paper is organized as follows: Section 2 presents background on ASP, product configuration, and the OOASP framework, as well as our running example and previous results regarding the interactive configurator. We then proceed to present our novel contributions. Section 3 contains detailed explanations about the four smart expansion functions and their implementation in our interactive configurator, and Section 4 introduces our novel *clinguin* UI. Finally, Section 5 demonstrates the experimental results of our new implementation, comparing its performance to the previous version, and Section 6 concludes this paper.

[1] https://potassco.org/clingo/

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

2 Background

2.1 Answer set programming

A logic program consists of rules of the form

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

where for $1 \leq i \leq o$, each $\texttt{a}_i$ is an atom of form $\texttt{p(t}_1\texttt{,...,t}_k\texttt{)}$ and all $\texttt{t}_j$ are terms (variables, constant terms, or function terms). For $0 \leq m \leq n \leq o$, atoms $\texttt{a}_1$ to $\texttt{a}_m$ are often called head atoms, while $\texttt{a}_{m+1}$ to $\texttt{a}_n$ and $\texttt{not a}_{n+1}$ to $\texttt{not a}_o$ are also referred to as positive and negative body literals, respectively. An expression is said to be ground if it contains no variables. As usual, `not` denotes (default) negation. A rule is called a fact if $m = n = o = 1$, normal if $m = 1$, and an integrity constraint if $m = 0$. In what follows, we deal with normal logic programs only, for which m is either 0 or 1. Semantically, a logic program induces a set of stable models, being distinguished models of the program determined by the stable models semantics (Gelfond and Lifschitz 1990).

To ease the use of ASP in practice, several extensions have been developed. Multi-shot solving, which is one such extension, allows for solving continuously changing logic programs in an operative way. In *clingo*, this can be controlled via an API for implementing reactive procedures that loop on grounding and solving while reacting, for instance, to outside changes or previous solving results.

We want to highlight here the use of so-called *assumptions* and *externals* (Kaminski *et al.* 2023). The former are added to the solving process and can be interpreted as integrity constraints that force the answer sets to contain certain atoms without providing evidence for them. The latter are specified by the `#external` directive and allow for declaring atoms whose truth value can be set via the API. This provides the tools to continuously assemble ground rules evolving at different stages of a reasoning process and to change program behavior by manipulating the truth values of atoms.

2.2 Product configuration and OOASP

Product configuration as an activity produces the specification of an artifact that is assembled from instances of given component types and that conforms to a given set of constraints between those components. Component types can have attributes, thus components can be parametrized. Furthermore, components are related via part-of, is-a, or other relationships (Felfernig *et al.* 2014). Many configuration problems are dynamic, meaning the number of necessary components for a solution is unknown in advance (Falkner *et al.* 2016).

OOASP (Falkner *et al.* 2015, 2018) is an ASP-based framework to encode and reason about object-oriented problems such as configuration problems. It defines a Domain Description Language (DDL) specific to the domain of object-oriented models that can be represented by a modeling language corresponding to a UML class diagram. OOASP-DDL defines ASP predicates to encode models (classes, subclass relations, associations, and attributes) and instantiations (instances, is-a relations, instance-level associations, and attribute values). Furthermore, it provides a uniform way to encode constraints.

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

Table 1. *OOASP-DDL predicates for the encoding of models*

Predicate	Description
`ooasp_class(C)`	`C` is a class
`ooasp_subclass(SubC,SupC)`	`SubC` is a subclass of `SupC`
`ooasp_assoc(A,C1,C1Min,C1Max,` `C2,C2Min,C2Max)`	In association `A`, each instance of class `C1` is associated to `C2Min-C2Max` instances of class `C2`, and each `C2` instance to `C1Min-C1Max` `C1` instances
`ooasp_attr(C,A,T)`	`A` is an attribute of class `C` with type `T`
`ooasp_attr_enum(C,A,D)`	`D` is an element of the domain of attribute `A` of class `C`

Table 2. *OOASP-DDL predicates for the encoding of instantiations*

Predicate	Description
`ooasp_isa(C,O)`	`O` is an object of class `C`
`ooasp_associated(A,O1,O2)`	Object `O1` is associated to object `O2` in association `A`
`ooasp_attr_value(A,O,V)`	The attribute `A` of object `O` has value `V`

Table 1 shows the OOASP-DDL predicates for the encoding of models, and Table 2 shows the OOASP-DDL predicates for the encoding of instantiations.[2]

OOASP constraints are defined using the predicate `ooasp_cv` (where "cv" stands for "constraint violation"). Rules with head atoms of this predicate are used instead of ASP constraints to enable configurations to be *checked*, that is, to derive which constraints are violated in a given configuration. To enforce a configuration to be consistent, the ASP constraint: `:- ooasp_cv(C,O,M,L)` is added, forbidding any constraint violations. An `ooasp_cv` atom contains four terms: a unique constraint identifier `C`, the identifier of the faulty object `O`, a string containing a message `M` describing the issue, and a list `L` of additional explanatory terms.

OOASP distinguishes integrity constraints from domain-specific constraints. The former are defined in the OOASP framework itself and refer to issues such as invalid values and violations of association cardinalities. Domain-specific constraints can be defined by a user of OOASP as additional rules that derive `ooasp_cv` atoms.

An instantiation (configuration) defined by the predicates from Table 2 is complete if every object is an instance of an instantiable class, and it is valid (consistent) if no constraint violations can be derived from it. We follow the convention that only leaf classes (i.e., classes that have no subclasses) are instantiable, so every object must be an instance of a leaf class in a complete configuration. Finding a complete and valid configuration is usually an interactive task, iteratively involving user interactions (decisions) and automatic reasoning by a solver, for example, an ASP solver (Falkner *et al.* 2020). The goal of our work is to support interactive configuration in a framework based on OOASP.

[2] We here present a version of OOASP-DDL that has evolved from the original definition (Falkner *et al.* 2015) and that has also been slightly simplified for this paper.

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

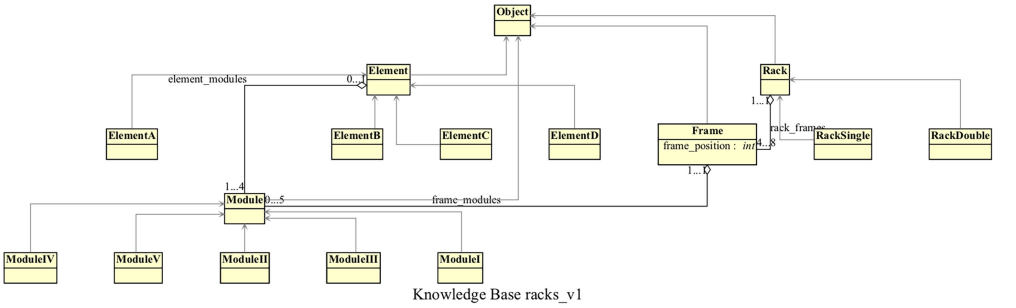

Fig. 1. Class diagram for the racks knowledge base generated by *clingraph*.

2.3 Running example

We use an extension of the typical hardware racks configuration problem (Falkner *et al.* 2015) as running example. The UML class diagram (Figure 1) shows all concepts and relations of the racks knowledge base. Additionally, it includes the following domain-specific constraints:

C1. An ElementA/B/C/D requires exactly 1/2/3/4 objects of type ModuleI/II/III/IV
C2. Instances of ModuleI/II/III/IV must be connected to exactly one Element
C3. A RackSingle/RackDouble has exactly 4/8 Frames
C4. A Frame containing a ModuleII must also contain exactly one ModuleV

This example captures the essence of a typical configuration knowledge base in an industrial setting. Of course, real-life industrial knowledge bases are much larger (>100 classes, associations, attributes). Another property of these knowledge bases is that the number of objects required for a solution is not known beforehand.

2.4 Interactive configurator

In our previous work (Comploi-Taupe *et al.* 2023),[3] we have identified and implemented eight distinct interactive tasks for product configuration. We have also provided an API as well as a UI implemented with *ipywidgets* and using *clingraph* to visualize the model and the configuration graph. The interactive configurator presented in this paper enables users to build a complete configuration incrementally based on the OOASP framework, by combining user actions with automatic solving. The implementation takes advantage of *clingo*'s multi-shot capabilities (Gebser *et al.* 2019), ensuring that rules are grounded dynamically as the configuration evolves. This approach avoids the need for re-grounding and utilizes learned constraints.

The system allows users to carry out edition tasks (T1–T4), which amount to setting the type of existing objects, adding or removing associations, setting values for attributes, and adding new objects. Additionally, three key reasoning tasks are integrated into the system: task T5 generates the configuration using choice rules, task T6 checks if the configuration violates any constraints, and task T7 provides the user with available edit options through brave reasoning. Tasks T6 and T7 employ two externals,

[3] Source code for this version can be found in https://github.com/siemens/OOASP/tree/v1.0.0

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

namely, `check_permanent_cv` and `check_potential_cv`, which help manage constraint violations. The potential violations are those that can be fixed later in the process, while permanent violations cannot be resolved. Finally, task T8 encapsulates the overall process, where a partial configuration ($\mathcal{P}$) is extended incrementally into a complete and consistent one ($\mathcal{C}$), by adding new objects until a complete and consistent configuration is found. This task was identified as the most expensive one in terms of performance, as it requires the solver to check for the existence of $\mathcal{C}$ with the current objects leading to multiple costly unsatisfiability checks before finding a solution.

3 Improved interactive configuration

In this section, we outline the new contributions that build upon the work presented by Comploi-Taupe *et al.* (2023) and whose main goal is to improve the performance and usability of the interactive configurator.

Let us first provide a novel formalization of task T8, which will be the focus of our improvements.

Definition 1.
*A **configuration problem** is defined by a tuple* $\langle C, AN, A, AtN, At, D, O, sat\rangle$, *where* C *is the set of instantiable classes,* AN *is a set of unique association names,* $A : AN \to C \times C$ *is a total function that defines the involved classes for each association,* AtN *is a set of attribute names,* $At \subseteq C \times AtN$ *is the set of attributes,* D *is the universal domain of attribute values,* O *is the (possibly infinite) set of potentially usable objects, and* sat *is a function that maps a partial configuration* $\mathcal{P}$ *as defined below to the value* $\top$ *iff* $\mathcal{P}$ *satisfies all the constraints imposed by the configuration problem.*[4]

Definition 2.
*A **(partial) configuration** $\mathcal{P}$ of a configuration problem as defined above is defined by a tuple* $\langle O_\mathcal{P}, instanceof_\mathcal{P}, assocs_\mathcal{P}, values_\mathcal{P}\rangle$, *where* $O_\mathcal{P} \subseteq O$ *is the set of instantiated objects,* $instanceof_\mathcal{P} : O \rightharpoonup C$ *is a partial function mapping instantiated objects to their classes,* $assocs_\mathcal{P} : AN \to 2^{O_\mathcal{P} \times O_\mathcal{P}}$ *is a function defining the instantiated associations, and* $values_\mathcal{P} : At \times O \rightharpoonup D$ *is a partial function mapping attributes to their values.*

A configuration is *complete* if every object is an instance of an instantiable class (i.e., the function *instanceof* is total) and every attribute has a value (i.e., for all $at = (c, a) \in At$ and all $o \in O_\mathcal{P}$ where $instanceof_\mathcal{P}(o) = c$, we get that $values_\mathcal{P}(at, o)$ has a value), and it is *consistent* if the constraints are satisfied (i.e., $sat(\mathcal{P}) = \top$).

The latter condition also enforces the lower bounds of all associations to be satisfied, and constraints may restrict the domains of attributes.

Task T8, extending a partial configuration into a complete and consistent one, can then be formally defined as follows: An incremental extension of a partial configuration to a complete and consistent one is a sequence of (partial) configurations $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n\rangle$, where for each $1 \leq i < n$, we get that $\mathcal{P}_{i+1}$ is a proper extension of $\mathcal{P}_i$; that is, it holds that $\mathcal{P}_i \neq \mathcal{P}_{i+1}$, $O_{\mathcal{P}_i} \subseteq O_{\mathcal{P}_{i+1}}$, $instanceof_{\mathcal{P}_i} \subseteq instanceof_{\mathcal{P}_{i+1}}$, $values_{\mathcal{P}_i} \subseteq values_{\mathcal{P}_{i+1}}$, and for all

[4] The detailed way how to define constraints is abstracted away here. Constraints include integrity constraints as well as domain-specific constraints as described in Section 2.2.

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

$an \in AN : assocs_{\mathcal{P}_i}(an) \subseteq assocs_{\mathcal{P}_{i+1}}(an)$. Furthermore, the configuration $\mathcal{P}_n$ is complete and consistent as defined above.

Our main contribution in this paper is the introduction of so-called *smart expansion functions*. For this purpose, we present four different functions that focus on improving the performance of task T8 by reducing the search space of the configuration problem. In the most basic implementation of task T8, objects are added incrementally to the configuration without any class restrictions or associations, leading to a combinatorial explosion of the search space. To mitigate this, we inspected new techniques that allow us to use existing information to infer additional knowledge about the required objects while also reducing the number of unsatisfiability checks.

We enhanced the usability of the system by extending the OOASP-DDL with the concept of association specializations (Taupe *et al.* 2016). This concept mirrors the idea of subclassing in OOASP but applies it to associations. With this extension, users can define associations specific to subclasses that override those of their parent classes. Therefore, domain-specific constraints C1–C3 from Section 2.3 can now be integrated into the knowledge base directly. For instance, by adding the facts `ooasp_assoc_specialization(element_modules1, element_modules)` and `ooasp_assoc(element_modules1, elementA, 1, 1, moduleI, 1, 1)`, we introduce a specialized version of the `element_modules` association, namely `element_modules1`, which enforces that each `elementA` must be associated with exactly one `moduleI`, and vice versa. However, constraint C4 cannot be expressed in this way and remains directly implemented in the ASP encoding. Overall, this extension increases the expressiveness of the language, reduces the need for domain-specific constraints, and enables the encoding to leverage this structural information during solving, as well as for the inferences in the smart functions.

Additionally, we also improved the performance of the system by removing the use of externals for user input and replacing them with assumptions, which was possible thanks to the reframing of task T6 (checking if $\mathcal{P}$ is complete or if it violates any constraints). Originally, this task needed to generate facts about which constraints were violated without solving the configuration. For this, it was necessary to disable all choice rules to avoid fixing the violations. However, since assumptions can be considered as constraints, using them to represent user input is only possible if the program has rules to obtain the assumed atoms, which requires the choice rules. We were able to refocus this in our new version by restricting feedback about constraint violations to use the smart functions described below. This change omits certain feedback details, such as missing attributes, but still provides information about missing objects and associations, which is arguably more relevant, at the benefit of removing the complexity introduced by externals.

We dedicate the remainder of this section to the illustration of our implementation of task T8 followed by a detailed explanation of the four smart expansion functions.

3.1 Smart functions

As mentioned in Section 2.4, our basic approach to solving the configuration problem incrementally consisted in adding objects to the configuration one by one and checking for unsatisfiability after each addition. Our novel contribution enhances this multi-shot

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

approach by adding four different smart expansion functions. The idea is that before checking for the existence of $\mathcal{C}$ with the current (grounded) objects, we apply the smart expansion functions, which determine and add necessary specific objects or associations to $\mathcal{P}$. This postpones costly unsatisfiability checks to the end of the process and thereby limits their number to a minimum. As a consequence, satisfiable instances might be processed by the smart expansion functions, but since they already meet the constraints, no new objects will be added. The overhead of this additional computation is minimal compared to the cost of checking unsatisfiability after adding each object.

The entire process is outlined in Algorithm 1, which takes a partial configuration $\mathcal{P}$ and iterates until it is extended into a complete configuration. It first sets the external `check_potential_cv` to false, allowing the smart expansion functions to analyze the current configuration. Then, it iterates over the smart functions, which are responsible for adding new objects or associations to the partial configuration. Selection and order of smart functions can be customized. If any of the functions add new objects or associations, the algorithm starts again with the first smart function. Once $\mathcal{P}$ can't be extended further by the smart functions, the external `check_potential_cv` is set to true, and the solver is called to check for unsatisfiability. If the configuration is still unsatisfiable, it means that the current set of objects is insufficient to satisfy the constraints and that the smart functions were not able to detect this. In this case, the algorithm adds a new object without a fixed type to the configuration. Throughout this process, the program is grounded incrementally whenever a new object is added, either by the smart functions or by the algorithm itself.

Algorithm 1 Smart Incremental Solving

```
Input: Partial configuration P
1  done ← False;
2  while not done do
3      Set external check_potential_cv to False;
4      foreach f ∈ smart_functions do
5          config_was_extended ← f(P);
6          if config_was_extended then
7              goto line 3;
8      Set external check_potential_cv to True;
9      done ← Solve();
10     if not done then
11         Add abstract object;
```

The smart expansion functions exploit knowledge about the configuration model using auxiliary predicates to gather relevant information about the current configuration state. These atoms are then used to determine necessary objects and associations by calculating cautious and brave consequences (intersection and union of stable models, respectively).

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

To ensure a satisfiable answer that provides this information, we ignore potential constraints by setting the external `check_potential_cv` to false, while the permanent constraints remain active, since we want to discard anything that cannot be fixed by further interaction with the system.

We now proceed to present each smart expansion function in detail. The encodings presented here are simplified versions that omit the additional argument used for incrementally grounded objects. For the complete encodings, we refer the reader to the source code.[5]

3.1.1 ObjectNeeded

The first smart expansion function detects when additional objects of a specific class are required, creates them, and associates them with an existing object.

To illustrate the workings of the function, we will consider the following example in the remainder of this section: The input partial configuration consists of one Rack r and one Frame f without associations. The model (Figure 1) specifies that a Rack must be associated with at least four Frames. This means that even if r gets associated with f, at least three more Frames must be added to satisfy the requirement.

This information is encoded using atoms of the form `assoc_needs_object(ID,A,X,C,DIR)`, as shown in Listing 1. This predicate indicates that the object `ID` requires *at least* `X` objects of class `C`, associated through `A` in the direction `DIR`. The rule relies on the predicate `lb_violation`, which detects violations of the lower bound for association `A`, where the required number of objects is `CMIN` and the current number is `N`. This violation is then used in the head of the rule to determine how many additional objects are needed by computing the range from 1 to the missing amount, specifically `CMIN-N`.

```
1  assoc_needs_object(ID,A,1..CMIN-N,C,D) :-
2          lb_violation(ID,_,(A,CMIN,N,C,D)).
```

Listing. 1. Encoding to obtain the `assoc_needs_object/6` predicate.

The smart function analyzes these atoms within the cautious consequences to determine the necessary objects and associations. Since these atoms appear in the cautious consequences, they hold true in all models, indicating that the constraints cannot be satisfied with the current set of objects. In our example, this analysis leads to the conclusion that at least three additional Frames must be associated with r (identified by ID 1), as captured by the following target atoms:

```
1  assoc_needs_object(1,rack_frames,1,frame,1),
2  assoc_needs_object(1,rack_frames,2,frame,1),
3  assoc_needs_object(1,rack_frames,3,frame,1).
```

We employ multiple atoms under the concept of "at least" and use model intersections to eliminate target atoms from models where possible associations with existing objects were not considered. In our example, since potential constraint violations are not enforced, there exists a model in which r and f are not associated, which obtains target atoms for $X \in \{1, 2, 3, 4\}$. However, there is another model where r and f are associated,

[5] https://github.com/siemens/OOASP/tree/v2.0.0

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

so the atom for $X = 4$ is absent. As a result, it is discarded during the intersection process. This technique significantly improves performance compared to our previous attempts, where a single atom and an optimization statement were used to minimize constraint violations.

As a result, the smart function then analyzes the target atoms, determines the maximum value of X, and grounds three additional Frames, directly associating them with object r through the `rack_frames` association.

3.1.2 GlobalUpperBoundGap

This smart function calculates required objects using upper bounds by summing up the number of existing instances of a class and verifying whether the upper bounds can be satisfied globally. This is implemented in the encoding in Listing 2, where the presence of the atom `global_ub_gap(C,N)` in the cautious consequences signifies the need to add at least `N` objects of type `C`.

This function is limited to "target associations," in which one side involves a single object. These associations are collected using the predicate `is_target_assoc/2`, which is incorporated into the rule. The predicate `ooasp_assoc_limit/6` is then used to retrieve the upper bound for the association from the model. Next, the `#count` aggregate is applied to count the number of objects for each class within the association. The final line ensures that the upper bound exceeds the number of objects already in the configuration and performs the necessary calculation to determine how many additional objects are needed.

For example, a partial configuration with one Rack and nine Frames includes the atom `global_ub_gap(rack,1)`. This is calculated from the fact that the upper bound for Racks in the `rack_frames` association is 8, while the number of Frames is 9. As a result, the current Racks are insufficient to associate with all the Frames. In this particular case, the smart function *ObjectNeeded* has no effect, as there exists a configuration where each Frame is associated with the Rack, meaning the intersection of violations is empty.

It is important to note that, since this is a global calculation, no instantiated objects are considered, and therefore, no specific associations can be added.

```
1 global_ub_gap(C2,1..((NUM1 + MAX - 1) / MAX)-NUM2) :-
2     is_target_assoc(A,DIR),
3     ooasp_assoc_limit(A,max,DIR2,C2,MAX,C1),
4     DIR2!=DIR,
5     #count { ID:ooasp_isa(C1,ID) } = NUM1,
6     #count { ID:ooasp_isa(C2,ID) } = NUM2,
7     NUM2 * MAX < NUM1.
```

Listing. 2. Encoding to obtain the `global_ub_gap/2` predicate.

3.1.3 GlobalLowerBoundGap

This function operates similarly to the *GlobalUpperBoundGap* function, but for lower bounds. In this case, atoms `global_lb_gap(C,N)` in the cautious consequences indicate the need to add at least `N` objects of type `C`.

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

For example, consider a scenario with 2 Racks and 4 Frames. Although locally, the Racks and Frames appear to have enough objects to satisfy the lower bound of the `rack_frames` association, the corresponding rule computes that the global lower bound for the two Racks is 8 Frames. Therefore, the atom `global_lb_gap(frame, 4)` indicates the need to ground at least four additional Frames to fulfill this gap.

For brevity, we omit the full encoding, which is quite similar to Listing 2.

3.1.4 AssociationPossible

The final smart function reduces the search space by identifying viable associations to complete the configuration through brave consequences. The inclusion of the atom `assoc_possible(A,ID1,ID2)` in the brave consequences signifies that the association `A` between objects `ID1` and `ID2` can (possibly) be added. These atoms are calculated in Listing 3.

The rule uses the predicate from the smart function *ObjectNeeded* to verify that both objects `ID1` and `ID2` require an association. Additionally, the predicate `potential_assoc/5` ensures that it is indeed possible to associate the objects. Finally, we confirm that the classes of the objects are already set, preventing the imposition of a class through the association.

While this approach may affect the completeness of the solution by limiting available options, it significantly speeds up the process of finding or estimating the minimal domain size. We can see this improvement in an example with 17 Frames, where adding these associations reduces the search space.

```
1 assoc_possible(A,ID1,ID2) :-
2         assoc_needs_object(ID1,A,1,_,_),
3         assoc_needs_object(ID2,A,1,_,_),
4         potential_assoc(A,ID1,ID2,C1,C2),
5         user(ooasp_isa(C1,ID1)),
6         user(ooasp_isa(C2,ID2)).
```

Listing. 3. Encoding to obtain the `assoc_possible/3` predicate.

4 User interface with clinguin

We have developed a new prototypical UI for smart interactive configuration, replacing the previous *ipywidgets*-based implementation by *clinguin* (Beiser *et al.* 2024). *clinguin* is a system for generating UIs in ASP. To achieve this, *clinguin* employs a set of dedicated predicates to define layout, style, and functionality of the interface, while handling user-triggered events. This streamlined design simplifies the specification of continuous user interactions within an ASP system, specifically *clingo*.

The built-in capacities of *clinguin* to add user input as assumptions, and to provide options and inferences via brave and cautious reasoning, allowed us to directly integrate tasks T1–T3, T5, and T7 without the need for additional code. Our prototype enhances usability with features such as saving/loading configurations and a *clear* button, while utilizing *clinguin*'s integration with *clingraph* (Hahn *et al.* 2024) to visualize and interact with the configuration graph directly. A snapshot of the UI is shown in Figure 2.

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

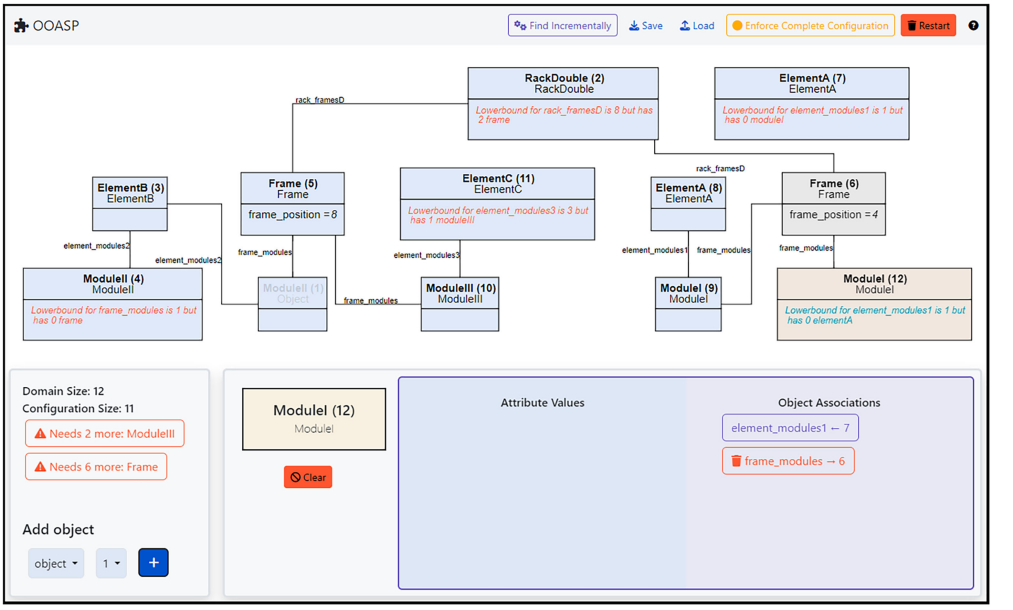

Fig. 2. Snapshot of prototype UI in *clinguin*.

5 Experimental results

The performance benchmarks were carried out for task T8 (completing a partial configuration) for 20 problem instances of increasing size and complexity.[6] Each of the instances is defined by a partial configuration containing the same number $n \in 1..20$ of objects of each Element subtype. The complete configuration obtained for the largest problem instance contains 325 objects. The performance testing was done on different permutations of the smart functions, with the resource consumption tracked by the *runsolver* tool (Roussel 2011). The testing was performed on a virtual machine with 32 GB RAM limit. Measures were taken to minimize external influence on the performance of the solver, for example, by disabling swapping.

Figures 3 and 4 visualize our main experimental results in terms of solving performance. We compare the old version of our implementation (Comploi-Taupe *et al.* 2023) with two different setups of the newest version. The curves in the figure represent the best-performing smart function permutations, with and without constraint C4 described in Section 2.3, in comparison to the old version. The smart expansion functions provide a significant improvement in terms of performance compared to the previous version in our benchmarks. The decrease in runtime and memory consumption allows the system to complete much more complex partial configurations than the previous version.

Constraints that cannot be expressed in the OOASP language (such as constraint C4), however, cannot always benefit from the smart functions. This is because the smart functions are not able to leverage the knowledge encoded by such constraints to derive missing objects or associations. The jumps in the runtimes in Figure 3 can be explained

[6] The benchmarking script is available as `benchmarks/ooasp_bench.sh` in https://github.com/siemens/OOASP/tree/v2.0.0.

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

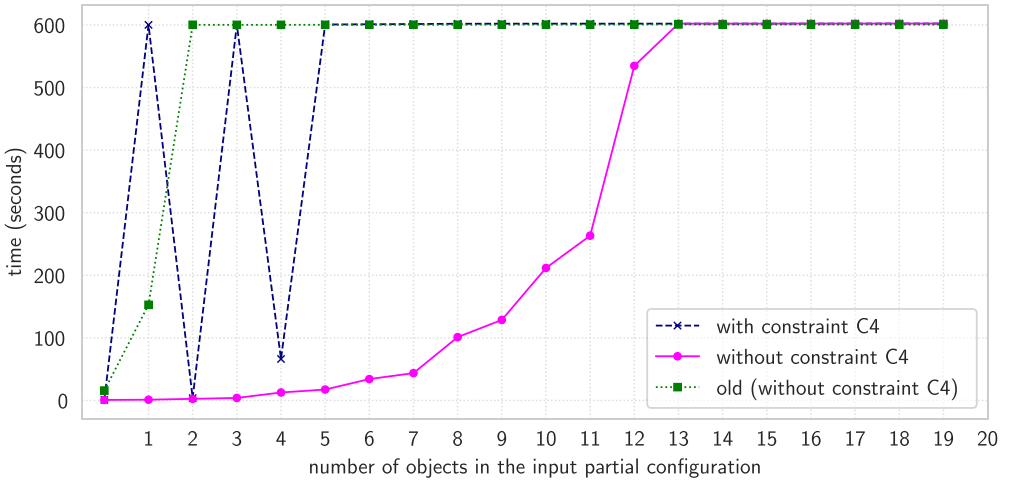

Fig. 3. Runtime (time-out: 600 s).

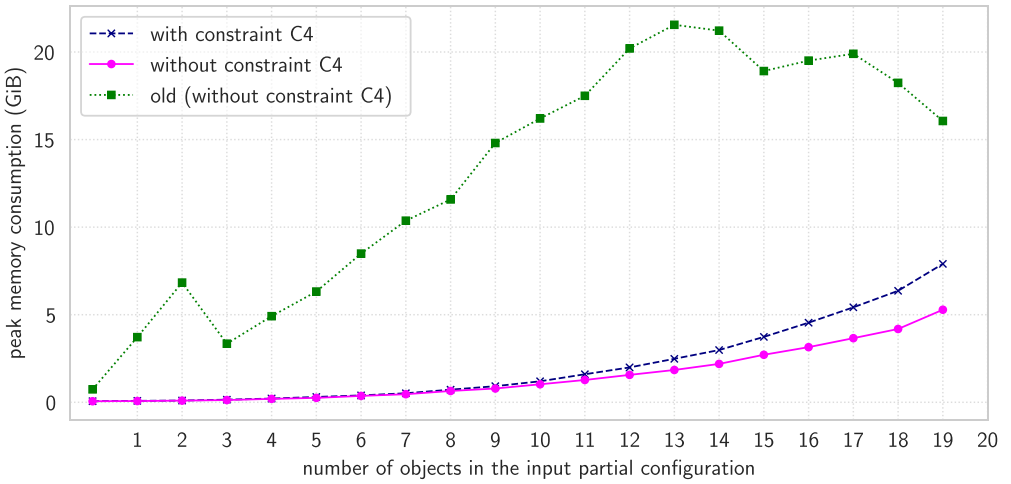

Fig. 4. Peak memory consumption.

by this fact and by observing that there are certain input (partial) configurations where the system is forced to derive generic objects (without any specific type) as the smart functions are not able to derive the specific types of needed objects.

Given the nature of the smart functions, there can be a huge disparity in the solver performance depending on the chosen order of application. Thus, the usage of smart functions does not immediately guarantee better performance. This variability, however, allows the combinations and the order of the smart functions to be tailored to a particular problem, for example, in portfolio solving (cf. Gebser *et al.* 2011a). We experimented with various permutations of at least three of the smart functions. The results show that the order of the smart functions has a significant impact on the performance such that some permutations can solve an instance in a matter of seconds, while others time out. On average, permutations of all four smart functions starting with *AssociationPossible*

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

provided the best results. In the results visualized in Figures 3 and 4, the permutation ⟨*AssociationPossible*, *ObjectNeeded*, *GlobalUpperBoundGap*, *GlobalLowerBoundGap*⟩ was used as it has been shown to have the best performance on the defined benchmarks.

6 Conclusions

When dealing with industrial configuration problems, the favored approach is often an interactive one, where the user is solving the problem in a step-wise manner while being guided through the configuration process. As these kinds of problems typically contain thousands of components, solving performance and the design of a clear, intuitive user experience are key. Our work addresses both of these issues by improving the performance of our existing interactive configurator (Comploi-Taupe *et al.* 2022, 2023) and by developing a new prototypical UI based on *clinguin*. While the former is achieved by introducing four new *smart expansion functions* to the ASP-based configuration API, the latter simplifies the specification of continuous user interactions within an ASP system, thereby facilitating future development of a user-friendly UI. The four smart expansion functions use cautious and brave reasoning to derive knowledge about the current configuration and to reduce the search space of the ASP solver. We demonstrated the working of the functions using our racks example as known from earlier work and compared the performance of the new functions with the old ones. Our experimental results show that the new functions are able to significantly reduce solving time and memory usage of the ASP solver *clingo*.

Future work should explore ways to generalize the smart functions such that they can be applied to configuration problems containing constraints not specified directly in the OOASP framework. We also plan to extend the UI prototype to support more complex configuration problems and improve the user experience.

Competing interests

The authors declare none.

References

Beiser, A., Hahn, S. AND Schaub, T. 2024. ASP-driven user–interaction with clinguin. In Technical Communications of the Fortieth International Conference on Logic Programming (ICLP'24), P. Cabalar and T. Swift, Eds. EPTCS, 215–228.

Comploi-Taupe, R., Falkner, A., Hahn, S., Schaub, T. AND Schenner, G. 2023. Interactive configuration with ASP multi-shot solving. In Proceedings of the Twenty-fifth International Configuration Workshop (CONF'23), J. Horcas, J. Galindo, R. Comploi-Taupe and L. Fuentes, Eds. Vol. 3509, 95–103, CEUR Workshop Proceedings. https://ceur-ws.org/Vol-3509/paper13.pdf.

Comploi-Taupe, R., Hahn, S., Schenner, G. AND Schaub, T. 2022. Challenges of developing an API for interactive configuration using ASP. In Proceedings of the Fifth Workshop on Trends and Applications of Answer Set Programming (TAASP'22), 2022, A. Tarzariol, F. Laferrière and Z. Saribatur, Eds. http://www.kr.tuwien.ac.at/events/taasp22/papers/TAASP_2022_paper_5.pdf.

https://doi.org/10.1017/S147106842510029X Published online by Cambridge University Press

Eiter, T., Faber, W., Fink, M. and Woltran, S. 2007. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence* 51, 2-4, 123–165. https://doi.org/10.1007/s10472-008-9086-5.

Falkner, A., Friedrich, G., Haselböck, A., Schenner, G. and Schreiner, H. 2016. Twenty-five years of successful application of constraint technologies at Siemens. *AI Magazine* 37, 4, 67–80. https://doi.org/10.1609/aimag.v37i4.2688.

Falkner, A., Friedrich, G., Schekotihin, K., Taupe, R. and Teppan, E. 2018. Industrial applications of answer set programming. *Künstliche Intelligenz* 32, 2-3, 165–176. https://doi.org/10.1007/s13218-018-0548-6.

Falkner, A., Haselböck, A., Krames, G., Schenner, G., Schreiner, H. and Taupe, R. 2020. Solver requirements for interactive configuration. *Journal of Universal Computer Science* 26, 3, 343–373. https://doi.org/10.3897/jucs.2020.019.

Falkner, A., Ryabokon, A., Schenner, G. and Shchekotykhin, K. 2015. OOASP: connecting object-oriented and logic programming, Proceedings of the Thirteenth International Conference On Logic Programming and Nonmonotonic Reasoning (LPNMR'15), F. Calimeri, G. Ianni and M. Truszczyński, Eds. Vol. 9345 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 332–345. https://doi.org/10.1007/978-3-319-23264-5_28.

Felfernig, A., Falkner, A., Atas, M., Erdeniz, S., Uran, C. and Azzoni, P. 2017. ASP-based knowledge representations for IoT configuration scenarios. In Proceedings of the Nineteenth International Configuration Workshop (CONF'17), L. Zhang and A. Haag, Eds. IESEG School of Management, 62–67. https://www.ieseg.fr/wp-content/uploads/2017/01/Proceedgins_Final.pdf#page=62.

Felfernig, A., Hotz, L., Bagley, C. and Tiihonen, J. 2014. *Knowledge-Based Configuration: From Research to Business Cases*. Elsevier/Morgan Kaufmann. https://doi.org/10.1016/C2011-0-69705-4.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1, 27–82. https://doi.org/10.1017/S1471068418000054.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M. and Ziller, S. 2011a. A portfolio solver for answer set programming: Preliminary report. In Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11), J. Delgrande and W. Faber, Eds. Vol. 6645 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 352–357. https://doi.org/10.1007/978-3-642-20895-9_40.

Gebser, M., Kaminski, R. and Schaub, T. 2011b. aspcud: A Linux package configuration tool based on answer set programming. In Proceedings of the Second International Workshop on Logics for Component Configuration (LoCoCo'11), C. Drescher, I. Lynce and R. Treinen, Eds. Vol. 65 of Electronic Proceedings in Theoretical Computer Science (EPTCS), 12–25. https://doi.org/10.4204/eptcs.65.2.

Gelfond, M. and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press.

Gelfond, M. and Lifschitz, V. 1990. Logic programs with classical negation. In Proceedings of the Seventh International Conference on Logic Programming (ICLP'90), D. Warren and P. Szeredi, Eds. MIT Press, 579–597.

Gençay, E., Schüller, P. and Erdem, E. 2019. Applications of non-monotonic reasoning to automotive product configuration using answer set programming. *Journal of Intelligent Manufacturing* 30, 1407–1422. https://doi.org/10.1007/s10845-017-1333-3.

Hahn, S., Sabuncu, O., Schaub, T. and Stolzmann, T. 2024. Clingraph: A system for ASP-based visualization. *Theory and Practice of Logic Programming* 24, 3, 533–559. https://doi.org/10.1017/S147106842400005X.

Kaminski, R., Romero, J., Schaub, T. and Wanko, P. 2023. How to build your own ASP-based system?!. *Theory and Practice of Logic Programming* 23, 1, 299–361. https://doi.org/10.1017/S1471068421000508.

Lifschitz, V. 2019. *Answer Set Programming*. Springer-Verlag. https://doi.org/10.1007/978-3-030-24658-7.

Roussel, O. 2011. Controlling a solver execution with the runsolver tool. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 4, 139–144. https://doi.org/10.3233/SAT190083.

Taupe, R., Falkner, A. A. and Schenner, G. 2016. Deriving tighter component cardinality bounds for product configuration. In 18th International Configuration Workshop, 2016. ISBN 979-10-91526-04-3. http://cp2016.a4cp.org/program/workshops/CWS-2016-Proceedings.pdf#section*.8.