Interactive Configuration with ASP Multi-Shot Solving

Richard Comploi-Taupe[1], Andreas Falkner[1], Susana Hahn[2,3], Torsten Schaub[2,3] and Gottfried Schenner[1]

[1]Siemens AG Österreich, Vienna, Austria
[2]University of Potsdam, Germany
[3]Potassco Solutions, Germany

Abstract

The area of product configuration has witnessed a growing demand for systems that can effectively guide users through the configuration process. These systems facilitate interactivity during configuration by combining user actions with automatic solving. In this paper, we present an API that fulfills the basic requirements of interactive configuration. Our implementation is based on the OOASP framework for object-oriented configuration in Answer Set Programming (ASP), leveraging multiple features of the ASP system *clingo* to dynamically introduce components.

1. Introduction

Product configuration has been one of the first successful applications of Answer Set Programming (ASP [1, 2]) [3]. Nonetheless, more than 20 years later, its use in product configurators is still challenging. One open challenge is to allow for interactivity during configuration.

Industrial product configuration deals with large problems. For example, even small infrastructure projects may contain thousands of components and hundreds of component types. Such configurations are typically solved step-by-step by combining interactive actions with automatic solving of sub-problems [4]. Configurator users, such as engineers and sales people, expect a system that guides them through the configuration process. Domain experts provide the configuration model that defines such a process and system.

Using a grounding-based formalism like ASP in this context introduces the risk of a grounding bottleneck [5] due to the large number of required components for satisfying all requirements. The required domain size can vary significantly and is not known beforehand, which leads to the necessity of dynamically introducing new components during the configuration process.

In this work, we present an Application Programming Interface (API) to satisfy basic requirements for interactive configuration [4]. Our implementation is based on OOASP [6], a framework for representing object-oriented configurations in ASP. Additionally, we exploit multiple features of the ASP system *clingo*[1] [7] to provide interactive functionalities.

After covering background on ASP, product configuration and OOASP, and on our running example in Section 2, we introduce our approach in detail in Section 3. The paper concludes with a discussion in Section 4.

2. Background

2.1. Answer set programming

A logic program consists of rules of the form

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

where each $\mathtt{a}_i$ is an atom of form $\mathtt{p(t}_1\mathtt{,}\ldots\mathtt{,t}_k\mathtt{)}$ and all $\mathtt{t}_i$ are terms, composed of function symbols and variables. For $1 \leq m \leq n \leq o$, atoms $\mathtt{a}_1$ to $\mathtt{a}_m$ are often called head atoms, while $\mathtt{a}_{m+1}$ to $\mathtt{a}_n$ and $\mathtt{not}\ \mathtt{a}_{n+1}$ to $\mathtt{not}\ \mathtt{a}_o$ 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 [8].

To ease the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include conditional literals and cardinality constraints [9]. The former are

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

ConfWS'23: 25th International Workshop on Configuration, Sep 6–7, 2023, Málaga, Spain
richard.taupe@siemens.com (R. Comploi-Taupe);
andreas.a.falkner@siemens.com (A. Falkner);
hahnmartinlu@uni-potsdam.de (S. Hahn);
torsten@cs.uni-potsdam.de (T. Schaub);
gottfried.schenner@siemens.com (G. Schenner)
0000-0001-7639-1616 (R. Comploi-Taupe); 0000-0002-2894-3284 (A. Falkner); 0000-0003-2622-2632 (S. Hahn); 0000-0002-7456-041X (T. Schaub); 0000-0003-0096-6780 (G. Schenner)

of the form[2] $\texttt{a}:\texttt{b}_1,\ldots,\texttt{b}_m$, the latter can be written as[3] $\texttt{s}\ \{\texttt{d}_1;\ldots;\texttt{d}_n\}\ \texttt{t}$, where a and $\texttt{b}_i$ are possibly negated (regular) literals and each $\texttt{d}_j$ is a conditional literal; s and t provide optional lower and upper bounds on the number of satisfied literals in the cardinality constraint. We refer to $\texttt{b}_1,\ldots,\texttt{b}_m$ as a condition. The practical value of both constructs becomes apparent when used with variables. For instance, a conditional literal like `a(X):b(X)` in a rule's body expands to the conjunction of all instances of `a(X)` for which the corresponding instance of `b(X)` holds. Similarly, `2 {a(X):b(X)} 4` is true whenever at least two and at most four instances of `a(X)` (subject to `b(X)`) are true. More sophisticated examples are given in Section 3.

A particular convenience feature are anonymous variables, denoted uniformly by an underscore '_'. Each underscore in a rule is interpreted as a fresh variable. In turn, atoms with anonymous variables are replaced by new atoms dropping these variables; the new atoms are then linked to the original ones by rules expressing projections.

Multi-shot solving 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. This is supported by two directives. First, a program can be partitioned into several subprograms by means of the directive `#program`; it comes with a name and an optional list of parameters. Such subprograms can then be grounded upon demand and added to the solver. Second, `#external` directives allow for declaring atoms whose truth value can be set via the API and/or rules that may be added later on. This allows us to continuously assemble ground rules evolving at different stages of a reasoning process and to change program behavior by manipulating the truth values of external atoms via the API.

Full details on the input language of *clingo* along with various examples can be found in the *Potassco User Guide* [10].

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 to each other via part-of or is-a relationships [11]. In most configuration problems, a dynamic number of components plays an important role [12].

OOASP[4] [6, 13] 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 modelling 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 (built-in and user-specific) constraints.

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.[5]

OOASP constraints are defined using the predicate `ooasp_cv` ("cv" stands for "constraint violation"). Rules with head atoms of this predicate are used instead of ASP constraints to enable configurations to be *checked*, i.e., to derive which constraints are violated in a given configuration (Listing 4). To enforce a configuration to be consistent, a simple ASP constraint forbidding `ooasp_cv` to be true can be added. An `ooasp_cv` atom contains four terms: a unique constraint identifier, the identifier of the faulty object, a string containing a message describing the issue, and a list 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 in the same format.

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 correct 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.

Configuration is usually an interactive task, iteratively involving user interactions (decisions) and automatic reasoning by a solver, e.g., an ASP solver [4]. The goal of our work is to support interactive configuration in a framework based on OOASP, because we think that its natural way of representing subclasses, parts hierarchies, rich element properties, and dynamically created configuration instances allows for understandable and precise product modeling.

[2] In rule bodies, they are terminated by '`;`' or '`.`' [10].

[3] More elaborate forms of aggregates are obtained by explicitly using function (e.g. `#count`) and relation symbols (e.g. `<=`) [10].

[4] https://github.com/siemens/OOASP

[5] We here present a version of OOASP-DDL that has already evolved from the original definition [6] and that has also been slightly simplified for this paper.

Table 1
OOASP-DDL predicates for the encoding of models

`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)`	A is an association in which each instance of the class C1 is associated to between C2Min and C2Max instances of class C2, and each instance of C2 is associated to between C1Min and C1Max instances of C1.
`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

`ooasp_isa(C,O)`	O is an object of class C
`ooasp_isa_leaf(C,O)`	O is an object of leaf 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

2.3. Running example

We use a typical hardware racks configuration problem as the running example for this paper. For easier comparison with non-incremental OOASP the running example is an extension of the racks configuration paper used in the original OOASP paper [6]. The UML class diagram (Figure 1) shows all concepts and relations of the racks knowledge base. This diagram was automatically generated by our Interactive API in integration with *clingraph* [14] using a visualization encoding.

Additionally to the constraints implied by the UML diagram, the following constraints hold for the domain:

- An ElementA/B/C/D requires exactly 1/2/3/4 objects of type ModuleI/II/III/IV
- Instances of ModuleI/II/III/IV must be required by exactly one Element
- A SingleRack/DoubleRack has exactly 4/8 Frames
- A Frame containing a ModuleII must also contain exactly one ModuleV

The running 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). And the constraints of the domain will vary considerable depending on additional requirements imposed by the customer, regulations, geographic location, etc. Notice that the knowledge base does not contain any restrictions on the number of objects in a configuration or on the order in which objects must be created.

Another property of these knowledge bases is that the number of objects required for a solution is not known beforehand. For example, suppose the user interactively created 5 objects of type ModuleI. The user could assign those modules to the same frame and assign the frame to a rack. Or the user could assign the modules to different frames and assign those to different racks. In any case, a rack must be connected to at least four frames. Therefore, the first configuration has 10 objects (5 modules, 4 frames, 1 rack), while the second, equally valid configuration has 30 objects (5 modules, 20 frames, 5 racks).

3. Interactive Configurator

Our Configuration API (CAPI) is implemented using Python, relying heavily on multiple features provided by *clingo*'s Python API, as well as the systems *clorm*[6] and *clingraph*. *Clorm* is a Python library providing an Object Relational Mapping (ORM) interface to *clingo*, which we use to map the OOASP predicates defining the knowledge base and the configuration into Python classes. These elements are then visualized as graphs (resembling UML diagrams) using *clingraph*. For interactive configuration, we created a scientific prototype User Interface (UI) using *ipywidgets* that employed our CAPI functionalities.

The basic idea behind our approach is to modularize the encodings so that the program can be built incrementally as the number of instantiated objects in the configuration increases based on user interaction. To that end, we use the multi-shot capabilities of *clingo* to solve these continuously changing logic programs. This approach avoids re-grounding and benefits from learned constraints by grounding and solving on demand. More specifically, we defined subprograms that depend on the identifier of each newly introduced object, namely `new_object`. Therefore, whenever the domain size is extended by a new object, all the rules referring to this object are grounded. In this sense, our implementation

[6] https://github.com/potassco/clorm

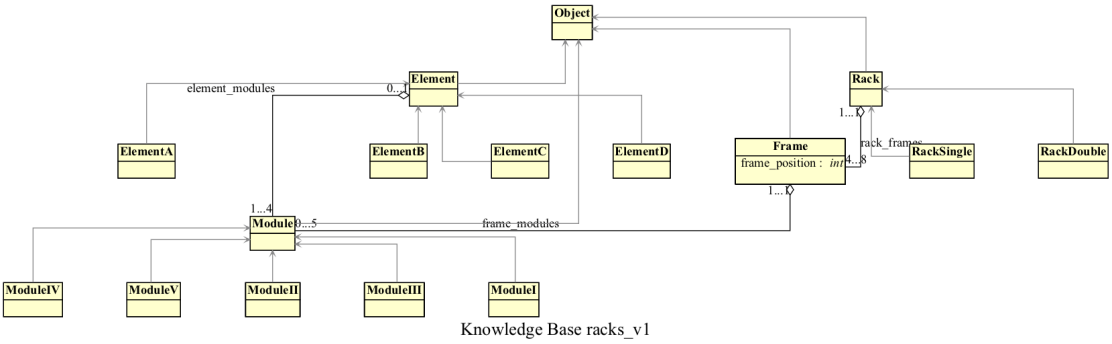


Figure 1: Class diagram for the racks knowledge base generated by *clingraph*.

differs from the previous work [15], in which subprograms were subject to domain-specific actions.

3.1. Interactive tasks

We introduce eight fundamental interactive tasks, derived from [4] and adapted to our multi-shot setting, allowing users to edit a partial configuration $\mathcal{C}_\mathcal{P}$ and construct a complete configuration $\mathcal{C}_\mathcal{C}$. First, the user can modify $\mathcal{C}_\mathcal{P}$ through the following interactive tasks:

T1. Setting and un-setting the type of an existing object.
T2. Adding and removing associations between two objects.
T3. Setting and un-setting values for attributes.

Such tasks are done using external atoms in *clingo*, so that no re-grounding is required. Due to lack of space, we will focus only on the encoding of task T3. Tasks T1 and T2 are encoded in a similar way. We show in Listing 1 how the user input is handled for task T3. Line 1 corresponds to *clingo*'s program directive indicating that the subprogram depends on `new_object`. Thus, all the following rules will be grounded on demand when a new object is introduced. Lines 2 and 3 define an external atom `user(ooasp_attr_value(A,new_object,V))` for each attribute `A` and value `V` of the `new_object`. Notice that we need to generate all possible combinations since the object can be assigned to any class. The truth value of these externals will be set based on the user's selection. Finally, the rule in Lines 4 and 5 makes sure that if the user selected a value for an attribute it will be considered in the encoding.

T4. Extending the configuration with a new object.

As mentioned before, the grounding of subprograms will exclusively occur when the user performs task T4. Consequently, in the rest of the tasks the number of objects will remain fixed. Note that the newly introduced

```
1  #program domain(new_object).
2  #external user(ooasp_attr_value(A,new_object,V)):
3          ooasp_attr_enum(_,A,V).
4  ooasp_attr_value(A,new_object,V) :-
5          user(ooasp_attr_value(A,new_object,V)).
```

Listing 1: User input

object will not have a type; its type will be set explicitly by the user with T1 or by the system with T5.

Finally, we identified three reasoning tasks in which solving is necessary.

T5. Using the current objects to generate $\mathcal{C}_\mathcal{C}$ from $\mathcal{C}_\mathcal{P}$ via choice rules.[7]
T6. Checking if $\mathcal{C}_\mathcal{P}$ is complete or if it violates any constraints.
T7. Obtaining the list of available edit-options for the user via brave reasoning.

These tasks are distinguished within the encoding via externals. The truth values of these externals is controlled internally depending on the task selected by the user. In this case, one external atom `guess` states that the guessing of objects' types, associations and values is active. Two additional externals `check_permanent_cv` and `check_potential_cv` activate the integrity constraints for the two types of constraints. Constraints are divided in this way, since we need to take into consideration that we are building $\mathcal{C}_\mathcal{C}$ in an interactive and incremental way, therefore some of these constraints might be violated on $\mathcal{C}_\mathcal{P}$ but fixed once $\mathcal{C}_\mathcal{C}$ is reached. The intuition for this decision can be taken from the fields of Runtime Verification and Monitoring, where a constraint is either *satisfied*, *potentially violated* (might or might not remain a violation in the future) or *permanently violated* (a violation in all possible futures). *Potential constraint violations* are those that can potentially be fixed by adding more information in a later stage of the process.

[7] A choice rule is a rule with a cardinality constraint in the head.

```
1  1 { ooasp_attr_value(A,new_object,V):
2                  ooasp_attr_enum(C,A,V) } 1 :-
3          ooasp_isa(C,new_object),
4          ooasp_attr(C,A,T),
5          ooasp_attr_enum(C,A,_),
6          guess.
```

Listing 2: Choice rule to guess the value of an attribute

```
1  :- ooasp_cv(CV,_,_,_),
2          not ooasp_potential_cv(CV),
3          check_permanent_cv.
4  :- ooasp_cv(CV,_,_,_),
5          ooasp_potential_cv(CV),
6          check_potential_cv.
```

Listing 3: Integrity constraints enforcing constraint violations

For instance, a lower bound of an association that has not been reached, or a value that is missing. These constraints are identified in the encoding with the predicate `ooasp_potential_cv`. On the other hand, *permanent constraint violations* refer to violations that can no longer be fixed, such as upper bounds of an association or an attribute value of a wrong type.

For task T5, `guess`, `check_permanent_cv` and `check_potential_cv` are set to true in order to find a complete and valid configuration $\mathcal{C}_\mathcal{C}$. This is achieved by using choice rules to generate possible values, types and associations for the objects, which are activated by the external `guess`. If no $\mathcal{C}_\mathcal{C}$ can be found with the current number of objects the result of the task will be UNSATISFIABLE. To illustrate this, Listing 2 shows the choice rule to select a value for an attribute. The rule can be read as follows: if the new object is of type C (Line 3), where type C has an attribute A (Line 4) with some elements in the domain (Line 5), and the guessing is active (Line 6), then out of all the possible values for A choose a single value V. Notice that this rule is also grounded incrementally and uses the corresponding `new_object`.

For task T6, we set the externals `check_permanent_cv` and `check_potential_cv` to false so that all the `ooasp_cv` atoms are part of the computed stable model, thus deriving the issues with $\mathcal{C}_\mathcal{P}$. In Listing 3 we show the integrity constraints handling the constraint violations, which are also grounded incrementally. Lines 1 to 3 make sure that no constraint violation is derived if the external `check_permanent_cv` is true and the constraint violation CV is not a potential but a permanent one. Similarly, the second constraint (Lines 4 to 6) enforces potential constraints when `check_potential_cv` is true.

For task T7, we want to provide the user with valid actions from T1, T2 and T3. To achieve this, potential constraints are ignored by setting the external `check_potential_cv` to false so that we allow constraints of this type to be violated in $\mathcal{C}_\mathcal{P}$ while still getting a satisfiable answer. However, the permanent constraints should remain active since we want to discard anything that can't be fixed by further interaction with the system. With this set, we use the brave reasoning capabilities of *clingo* to obtain the union of all stable models, and thus, all the possible options for types, values of attributes, and associations.

```
1  ooasp_potential_cv(no_val).
2  ooasp_cv(no_val,new_object,"Missing value for {}",(A,)) :-
3          ooasp_attr(C,A,T),
4          ooasp_attr_enum(C,A,_),
5          ooasp_isa(C,new_object),
6          not ooasp_attr_value(A,new_object,_).
```

Listing 4: Constraint violation of a missing value

As before, we use the attribute values to exemplify the use of task T7 in Listing 4. The rule in Lines 2 to 6 derives the constraint violation `no_val` of having no value set for an attribute. As expected, this is a potential constraint (expressed in Line 1) since the user can later on select the missing value. The constraint violation is then derived for any attribute of the `new_object` that has no corresponding value assigned via `ooasp_attr_value`.

Some other checks might depend on values that have to be recomputed on every grounding step, such as the arity of an association. In other words, if an aggregate `#count` is used to compute the objects associated to `new_object` , it will only count the objects that are already grounded at that time. This means that the arity computed in previous steps must be disregarded. Therefore, we need an additional external `active(new_object)` that indicates the current step to know if the aggregate's value is older and thus expired. Notice that the current step corresponds to the object identifier that is being grounded at that time.

T8. Extend $\mathcal{C}_\mathcal{P}$ incrementally to generate $\mathcal{C}_\mathcal{C}$

Given all these functionalities, finding the smallest $\mathcal{C}_\mathcal{C}$ that extends $\mathcal{C}_\mathcal{P}$ can be encapsulated into the combined task T8. The program for task T8 will proceed following an incremental approach: $\mathcal{C}_\mathcal{P}$ is extended with a new object (T4) and then tries to generate $\mathcal{C}_\mathcal{C}$ (T5), these steps are repeated until a $\mathcal{C}_\mathcal{C}$ is found.

3.2. Performance

Knowing about the huge solution space, we improved efficiency right from the beginning by including symmetry breaking constraints which get rid of multiple symmetric configurations. The two symmetries identified can be found in Listing 5. Both symmetries correspond to permutations of the classes assigned to objects. The constraint in Lines 1 to 6 ensures that the classes assigned to objects smaller than the `new_object`

```
1  :- ooasp_isa_leaf(C1,new_object),
2        ooasp_isa_leaf(C2,ID),
3        ID<new_object,
4        C1<C2,
5        not user(ooasp_isa_leaf(C1,new_object)),
6        not user(ooasp_isa_leaf(C2,ID)).

8  :- ooasp_isa_leaf(_,new_object),
9        not ooasp_isa_leaf(_,ID),
10       ooasp_isa(_,ID),
11       ID<new_object.
```

Listing 5: Symmetry breaking constraints

are also smaller. Notice that this is only applied to decisions made by the solver, excluding assignments made by the user (Lines 5 and 6). Otherwise the user setting the class of an object (via T1) could lead to unsatisfiability. The constraint in Lines 8 to 11 makes sure that any objects left out from the configuration (with no class assigned) are always those with larger ids. For instance, the assignment $\{(1, C_1), (2, \textit{undef}), (3, C_2)\}$ would be removed by the second constraint in favor of $\{(1, C_1), (2, C_2), (3, \textit{undef})\}$. Similarly, $\{(1, C_1), (2, C_2), (3, C_1)\}$ would not be valid due to the first constraint, keeping the symmetric assignment $\{(1, C_1), (2, C_1), (3, C_2)\}$.

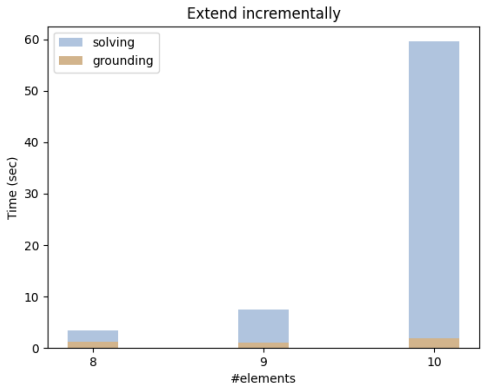
We performed some empirical tests in the system to check the performance based on the running example from Section 2.3. The aim of the first test was to generate a $\mathcal{C}_\mathcal{C}$ of size 41 with one RackSingle associated to four Frames, each Frame with four associated Modules with one corresponding Element. First, we extended $\mathcal{C}_\mathcal{P}$ with 41 objects via T4 where 18 of those objects were selected to be of the Element class. Then we used task T5 to find our expected $\mathcal{C}_\mathcal{C}$. Overall, these steps took 7 seconds of grounding time and 3 seconds of solving. At this point, any of the tasks T1, T2, T3, and T6 can be done without delay. However, obtaining the list of options with task T7 didn't finish within 5 minutes. This happened since the number of valid options is quite large when a user has 23 objects without an associated class. In practice we expect this to decrease with a more tightly defined $\mathcal{C}_\mathcal{P}$ during the interaction. As a second test, we analyzed task T8 by creating n Element instances and finding incrementally the corresponding $\mathcal{C}_\mathcal{C}$. The results can be found in Figure 2a for $n \in \{8, 9, 10\}$. When we increased n to 11, the task didn't finish within a 5 minute time out. Looking closely at the performance for $n = 9$ in Figure 2b, we can see that the issue lies on having to prove unsatisfiability for domain sizes 9 to 22. Proving unsatisfiability implies going trough all the search space to make sure there is no answer, which is quite costly as the domain increases. In our test, this is the case when trying to find a non-existing $\mathcal{C}_\mathcal{C}$ with a domain size of 22 objects. This was not the case in our previous test where we have all the required objects to obtain a satisfiable answer in a single call, comparable to the solving time taken for domain size 23 in Figure 2b.

3.3. User Interface Prototype

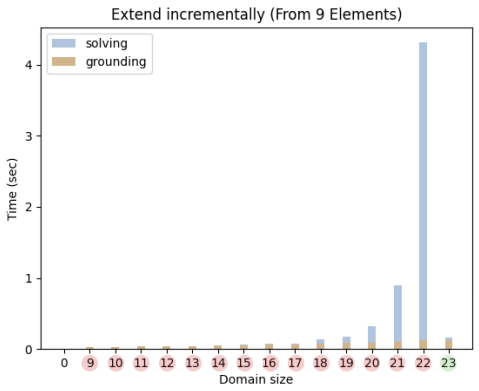
To give an impression of our prototype, we include screenshots (Figures 3,4) of the UI rendered in a Jupyter notebook for the racks examples from Section 2.3. This UI has 6 sections. The section on the upper left corner shows the current $\mathcal{C}_\mathcal{P}$ using *clingraph*. To the right, the history of actions taken by the user is rendered as a list in the **History** section. In our example, the user started with the first two actions being the extension of the domain by two new objects. This is done via the button in the **Extend** section below, corresponding to task T4. Then, in steps 3 and 4, the user assigned classes to objects 1 and 2, respectively, using the **Edit** section. This section employs T7 to generate a dropdown for each object with the list of possible options (T1, T2, T3). Step 5 corresponds to T6, triggered by clicking the button in the **Check** section. This action prints in red the constraint violations found for each object. In this case, Object 2 (the frame) violates the missing value constraint from Listing 4 and the lower-bound constraint, since it should be associated with one rack. Similarly, Object 1 (the instance of RackSingle) is violating the lower-bound constraint, as well as a domain-specific constraint enforcing it to be associated to exactly 4 frames. For the last task, namely T5, the user must work on the **Browse** section of the UI. By clicking the button labeled "Next solution", the system would perform task T5. However, since there is no $\mathcal{C}_\mathcal{C}$ with the current number of objects it would get an error. As mentioned before, this is overcome by extending $\mathcal{C}_\mathcal{P}$ incrementally, which is done by task T8 in step 6 when the user clicks on "Find incrementally" (Figure 4). As a consequence, the system will internally find $\mathcal{C}_\mathcal{C}$ and render it in the bottom right. In this *clingraph* image, the values from $\mathcal{C}_\mathcal{P}$ will appear in green. As a next step, the user could either browse through all the possible $\mathcal{C}_\mathcal{C}$ of the same size, or select the current $\mathcal{C}_\mathcal{C}$ as the new $\mathcal{C}_\mathcal{P}$ to be to be further edited. Notice that while browsing, no options are shown in the **Edit** section. This happens since we have a single *clingo* control object which is currently in the middle of solving, thus, it can't generate the brave consequences of T7.

4. Discussion

In this paper, we have presented an interactive configurator that enables engineers and salespeople, among other configurator users, to incrementally build configurations. Our contribution involves the development of an API built upon the OOASP framework and the *clingo* system. The OOASP framework provided us with a domain



(a) Times starting from 8, 9 and 10 Elements.



(b) The times for each call to task T5 for the given domain size, starting from 9 Elements.

Figure 2: Times for task T8 (Extending incrementally)

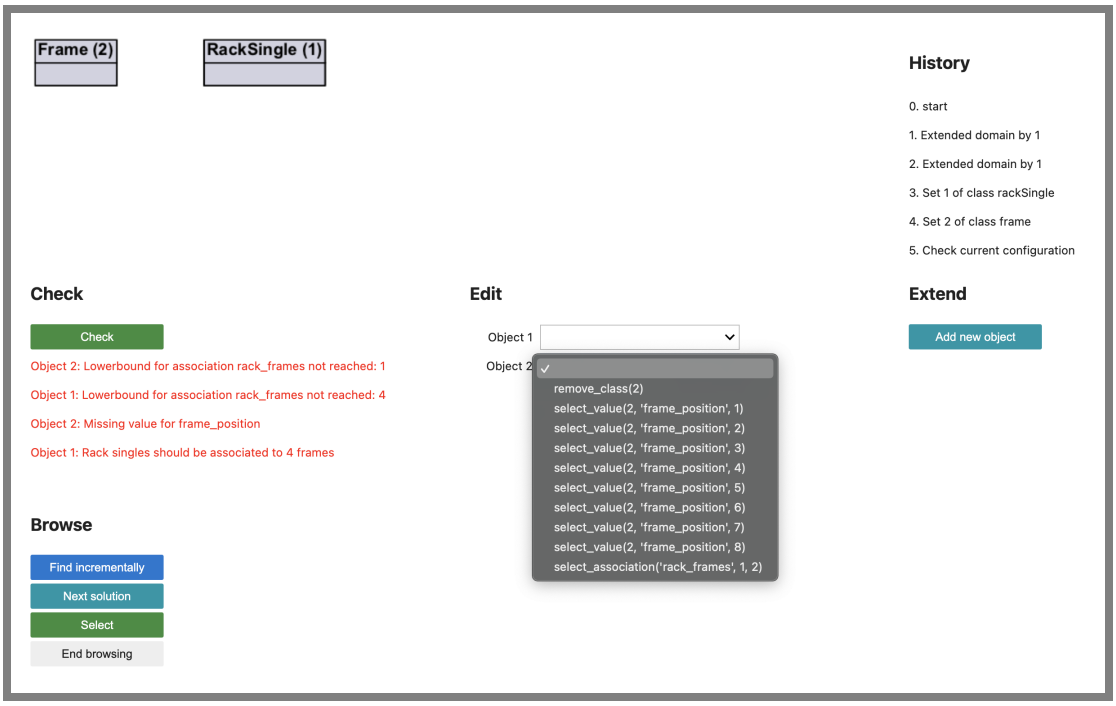


Figure 3: UI for interactive configuration generated with *ipywidgets* rendered in a Jupyter notebook.

description language to encode models, instances, and solutions, while *clingo*'s multi-shot capabilities allowed us to dynamically extend the configuration by adding components on demand. The integration of this multi-shot approach to interactive configuration distinguishes our approach from previous work.

To fulfill the basic requirements of interactive configuration, we identified and implemented eight distinct tasks, which we described in detail. To demonstrate the functionality of our API, we developed a prototype user interface and showcased the step-by-step creation of a configuration using our running example. As the UI is

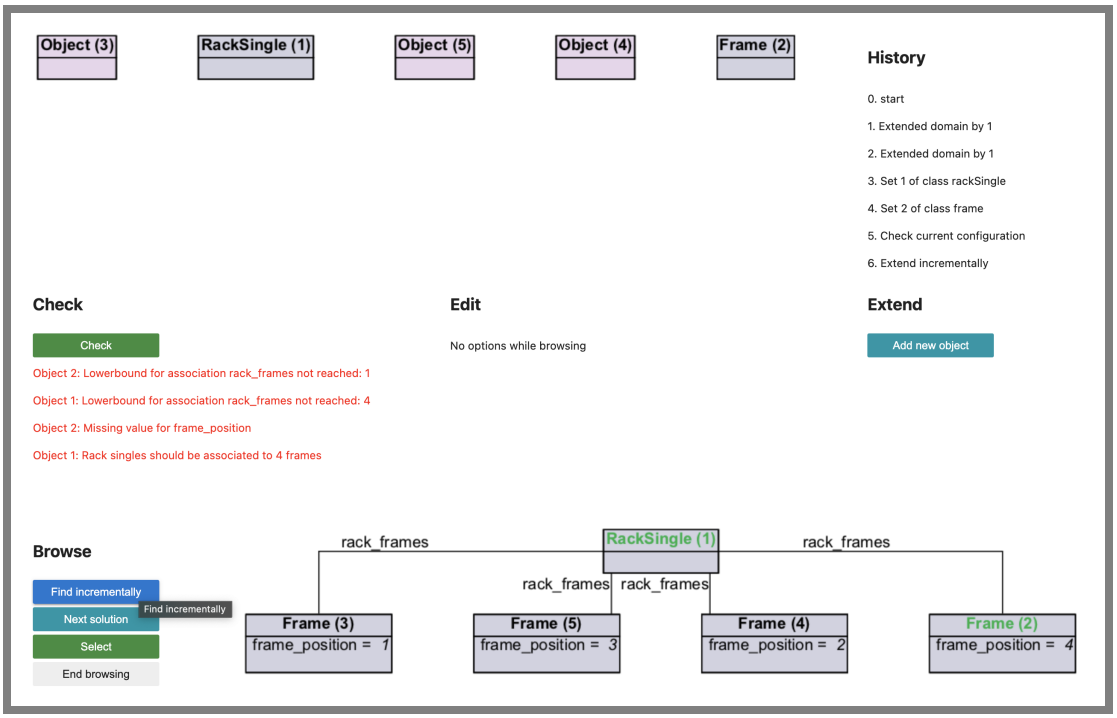


Figure 4: UI for interactive configuration generated with *ipywidgets* rendered in a Jupyter notebook. Where the user is browsing the extensions for $\mathcal{C}_{\mathcal{P}}$ into $\mathcal{C}_{\mathcal{C}}$.

currently in a prototypical stage, gathering real user feedback remains a future goal for subsequent versions of the system. To assess the performance of the system, we conducted empirical tests and identified the need for further improvements. In particular, we boosted efficiency by incorporating symmetry-breaking constraints.

However, we also encountered performance issues with our incremental approach when dealing with larger instances, which warrants future research. More specifically, we plan to explore alternative methods for extending the configuration beyond the one-by-one incremental process. For instance, we are interested in investigating scheduling techniques [16] and pre-computing the minimal number of required objects [17]. Additionally, we aim to enhance usability by incorporating additional advanced features, such as linear domains, which enable more sophisticated reasoning in the configuration process.

References

[1] V. Lifschitz, Answer Set Programming, 2019. doi:10.1007/978-3-030-24658-7.

[2] M. Gelfond, Y. Kahl, Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach, Cambridge University Press, New York, NY, USA, 2014.

[3] T. Soininen, I. Niemelä, J. Tiihonen, R. Sulonen, Representing configuration knowledge with weight constraint rules., in: A. Provetti, T. Son (Eds.), Proceedings of the AAAI Spring Symposium on Answer Set Programming (ASP'01), AAAI/MIT Press, 2001, pp. 195–201. URL: http://www.cs.nmsu.edu/%7Etson/ASP2001/20.ps.

[4] A. Falkner, A. Haselböck, G. Krames, G. Schenner, H. Schreiner, R. Taupe, Solver requirements for interactive configuration. 26 (2020) 343–373. doi:10.3897/jucs.2020.019.

[5] T. Eiter, W. Faber, M. Fink, S. Woltran, Complexity results for answer set programming with bounded predicate arities and implications, Ann. Math. Artif. Intell. 51 (2007) 123–165. doi:10.1007/s10472-008-9086-5.

[6] A. Falkner, A. Ryabokon, G. Schenner, K. Shchekotykhin, OOASP: connecting object-oriented and logic programming, in: F. Calimeri, G. Ianni, M. Truszczyński (Eds.), Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15), volume 9345 of *Lecture Notes in Artificial Intelli-*

gence, Springer-Verlag, 2015, pp. 332–345. doi:10.1007/978-3-319-23264-5_28.

[7] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82. doi:10.1017/S1471068418000054.

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

[9] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artificial Intelligence 138 (2002) 181–234.

[10] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, Potassco User Guide, 2 ed., University of Potsdam, 2015. URL: http://potassco.org.

[11] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen (Eds.), Knowledge-Based Configuration – From Research to Business Cases, Morgan Kaufmann, Boston, 2014. doi:10.1016/B978-0-12-415817-7.00029-3.

[12] A. A. Falkner, G. Friedrich, A. Haselböck, G. Schenner, H. Schreiner, Twenty-five years of successful application of constraint technologies at siemens, AI Mag. 37 (2016) 67–80. doi:10.1609/aimag.v37i4.2688.

[13] A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E. Teppan, Industrial applications of answer set programming, Künstliche Intelligenz 32 (2018) 165–176. doi:10.1007/s13218-018-0548-6.

[14] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, clingraph: ASP-based visualization, in: G. Gottlob, D. Inclezan, M. Maratea (Eds.), Proceedings of the Sixteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22), volume 13416 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 2022, pp. 401–414. doi:10.1007/978-3-031-15707-3_31.

[15] R. Comploi-Taupe, G. Francescutto, G. Schenner, Applying incremental answer set solving to product configuration, in: Proceedings of the 26th ACM International Systems and Software Product Line Conference – Volume B, Association for Computing Machinery, New York, NY, USA, 2022, pp. 150–155. doi:10.1145/3503229.3547069.

[16] Y. Dimopoulos, M. Gebser, P. Lühne, J. Romero, T. Schaub, plasp 3: Towards effective ASP planning, in: M. Balduccini, T. Janhunen (Eds.), Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17), volume 10377 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 2017, pp. 286–300. doi:10.1007/978-3-319-61660-5_26.

[17] M. Aschinger, C. Drescher, G. Gottlob, H. Vollmer, Loco – A logic for configuration problems, ACM Trans. Comput. Log. 15 (2014) 20:1–20:25. doi:10.1145/2629454.