Solving Object-oriented Configuration Scenarios with ASP *

Gottfried Schenner and **Andreas Falkner**
Siemens AG Österreich, Vienna, Austria
gottfried.schenner@siemens.com
andreas.a.falkner@siemens.com

Anna Ryabokon and **Gerhard Friedrich**
Universität Klagenfurt, Austria
anna.ryabokon@aau.at
gerhard.friedrich@aau.at

Abstract

The main configuration scenarios occurring in the domain of technical products and systems are consistency checking, completing a partial configuration, reconfiguration of an inconsistent configuration and finding the best knowledge base for future reconfigurations. This paper presents OOASP - a framework for the description of object-oriented product configurators using answer set programming and shows that it is able to solve the different (re)configuration scenarios occurring in practice. Thus, it is a step forward to close the conceptual gap between logic-based and object-oriented approaches for product configuration.

1 Introduction

A configurator is a software system that enables the user to design complex technical systems or services based on a predefined set of components. In modern configuration systems, domain knowledge - comprising configuration requirements (product variability) and customer requirements - is expressed in terms of component types and relations between them. Each type is characterized by a set of attributes which specify the functional and technical properties of real-world and abstract components of the configurable product. An attribute takes values from within a predefined domain. Furthermore, components are related/connected to each other in various ways. Each component type has a number of ports which allow to connect a component of that type with other components. A possible connection between two component types is modeled as a relation and its cardinality expresses the number of components that can be connected to a port. In most cases, modeling languages used in configuration allow to specify relations of the following types: classification (is-a), composition (part-of), association (user defined relations).

For simple customer products, a configuration system aims at finding a consistent and complete configuration for a given set of customer requirements and reconfiguration is seldom an issue. Reconfiguration occurs during the maintenance of technical systems with a long life-span, where parts of existing configurations have to be adapted continuously.

Reconfiguration is especially challenging if an existing system has to be extended with new functionality that was not part of the original system design. In this case, some relations between new and existing components have to be created or some of the existing relations have to be changed in order to meet modified configuration requirements. [Falkner and Haselböck, 2013] discuss typical problems occurring when configuration requirements are changed. Finding the best design for future configurations is an important task for the reconfiguration scenario since it allows to reduce costs during the production process.

In the current paper we present a generic configurator which uses an object-oriented approach to encode its knowledge base. In order to compute configurations the system uses answer set programming (ASP). We illustrate the mapping from an object-oriented formalism (UML) to logical descriptions using a simplified real-world example from Siemens. Additionally, the paper provides different insights on (re)configuration scenarios such as checking and completing a configuration, reconciliation and choosing the best knowledge base for reconciliation. Finally, we discuss challenges which frequently occur in practice and should be taken into account while solving (re)configuration problems.

The remainder of this paper is organized as follows: In Section 2 we introduce a sample configuration problem used as example throughout the paper. After an ASP overview in Section 3, we describe in Section 4 how object-oriented knowledge bases can be specified using ASP. In Section 5 various product configuration scenarios are discussed. Section 6 provides some evaluation details and in Section 7 we conclude.

2 Configuration example

Modules example is a simple hardware configuration problem. Figure 1 shows the configurable objects of the example domain in a UML diagram: hardware frames contain up to five modules of various types (A, B) and elements of various types are assigned to the modules (one by one). Additionally to the cardinality constraints implied by the UML diagram there are the following domain-specific constraints:

- Elements of type ElementA require a module of type ModuleA

*This work has been developed within the scope of the project RECONCILE (reconciling legacy instances with changed ontologies) and was funded by FFG FIT-IT (grant number 825071).

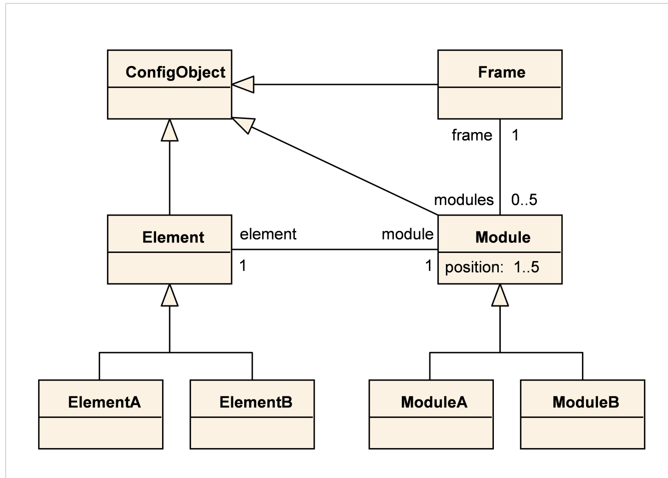

Figure 1: UML diagram for the modules example

- Elements of type ElementB require a module of type ModuleB
- The position of the modules of a frame must be unique

In a typical configurator scenario for this domain, a user creates a partial configuration consisting of elements of different types. Then the configurator extends the configuration by creating missing modules for the elements and by creating missing frames and assigning the modules to them. If the user manipulates a completed configuration, for instance by adding or removing elements, the configurator can restore consistency through reconfiguration, usually by keeping as much of the existing structure of the configured system as possible.

3 Answer set programming overview

Answer set programming is an approach to declarative problem solving which has its roots in logic programming and deductive databases. It is a decidable fragment of first-order logic interpreted under stable model semantics and extended with default negation, aggregation, and optimization. ASP allows modeling of a variety of (combinatorial) search and optimization problems in a declarative way using model-based problem specification methodology (see e.g. [Gelfond and Lifschitz, 1988; Eiter *et al.*, 2009; Brewka *et al.*, 2011] for details). ASP has a long history of being used for product configuration [Soininen and Niemel, 1998].

An ASP program is a finite set of rules of the form:

$$\mathtt{a} \,\mathtt{:-}\, \mathtt{b}_1, \ldots, \mathtt{b}_m, \mathtt{not}\ \mathtt{c}_1, \ldots, \mathtt{not}\ \mathtt{c}_k. \quad (1)$$

where $\mathtt{a}$, $\mathtt{b_i}$, and $\mathtt{c_j}$ are *atoms* of the form $\mathtt{predicate(term_1, \ldots, term_n)}$. A *term* is either a variable or a constant.

In most of ASP languages, variables are denoted by strings starting with uppercase letters and constants as well as predicates by strings starting with lower case letters. An atom together with its negation is called *literal*, e.g. $\mathtt{a}$ is a positive and $\mathtt{not\ a}$ is a negative literal. In the rule (1), the literal $\mathtt{a}$ is the *head* of the rule and the conjunction $\mathtt{b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_k}$ is the *body*. A rule with an empty head, standing for *false*, is called an *integrity constraint*, i.e. every interpretation that satisfies the body of the constraint is not an answer set (configuration solution). A rule with an empty body is called a *fact*. Rule (1) derives that the atom $\mathtt{a}$ in the head of the rule is *true* if all literals of the body are *true*, i.e. there is a derivation for each positive literal $\mathtt{b_1, \ldots, b_m}$ whereas none of the atoms of the negative literals $\mathtt{not\ c_1, \ldots, not\ c_k}$ can be derived.

Processing of a general ASP program P, in which atoms can contain variables, is done in two stages [Brewka *et al.*, 2011]. First the program is grounded, i.e. P is replaced by a possibly small equivalent propositional program $grnd(P)$ in which all atoms are variable-free. In the second stage an ASP solver is used to identify answer sets. Following the definition of configuration problems based on logical descriptions, presented in [Soininen *et al.*, 2001; Felfernig *et al.*, 2004], each configuration is a subset of a finite Herbrand-model. Given the stable model semantics used in ASP, a Herbrand interpretation I is a *model* of a program P *iff* (a) it satisfies all the rules in P, (b) for every atom $a_i \in I$ there exists a justification based on given facts and (c) I is minimal under set inclusion among all (consistent) interpretations.

In this paper we use an ASP dialect implemented in Gringo [Gebser *et al.*, 2011][1] which includes a number of extensions simplifying the presentation of the programs. Thus, it allows definition of *weight* constraints which are defined as $\mathtt{l[a_1 = w_1, \ldots, a_n = w_n]u}$ where $\mathtt{a_i}$ are atoms, $\mathtt{w_i}$ are weights of the atoms and $\mathtt{l, u}$ are integers specifying lower and upper bounds. Such constraints allow declaration of choices, i.e. such number of atoms from the set $\{\mathtt{a_1, \ldots, a_n}\}$ must be true that the sum of corresponding weights is between $\mathtt{l}$ and $\mathtt{u}$. If the lower or upper bounds are missing, then the ASP grounder substitutes $\mathtt{l} = \mathtt{0}$ and $\mathtt{u} = \mathtt{n}$, where $\mathtt{n}$ is the sum of the weights of all atoms in the set. A special case of the weight constraints are *cardinality* constraints where each weight $\mathtt{w_i} = \mathtt{1}$. Cardinality constraints are denoted by curly brackets.

ASP dialects include operators that are used for generating sets of atoms: The *range* operator ("..") is used to generate a set of atoms such that each atom includes one of the integer constants from a given range of integers. The *generate* operator (":") is used in weight constraints to create sets of atoms used in it.

Example Assume that we want to encode a simple problem instance of the modules example including two frames with ids 1 and 2, and six modules with ids ranging from 10 to 15. These customer requirements can be represented as facts:

```
frame(1..2). module(10..15).
```

The relation between modules and frames, i.e. that each module must be placed in exactly one frame, is encoded using a choice rule:

```
1{mod2fr(X,Y) : frame(Y)}1 :- module(X).
```

[1] Potassco ASP suite: http://potassco.sourceforge.net

When the rule above is grounded, the grounder generates six rules - one for each module. E.g., for module 10 it outputs:

```
1{mod2fr(10,1), mod2fr(10,2)}1 :- module(10).
```

In order to allow at most five modules to be put in a frame we add the following cardinality constraint to our program:

```
:- frame(X), 6 {mod2fr(Y,X) : module(Y)}.
```

Due to the cardinality constraint every configuration (answer set) containing a frame with more than 5 modules will be eliminated.

Identification of the preferred configuration solution can be done using the built-in optimization functionality of ASP solvers. In the ASP dialect used in this paper, the optimization is defined on a weighted set of true atoms and indicated via `#minimize` or `#maximize` statements.

4 OOASP framework

OOASP[2] is a framework for describing object-oriented knowledge bases in ASP. A knowledge base consists of the object model of the configurator and additional constraints which a valid configuration must satisfy. It is assumed that the object model of the object-oriented configurator can be described by an UML class diagram [Rumbaugh *et al.*, 2005]. The structure of a knowledge base and configurations are described by special ASP facts. This fact-based language can be seen as a domain specific language (DSL) for defining object-oriented knowledge bases and configurations on top of ASP. The DSL can represent multiple configurator knowledge bases and solutions in one ASP program. The OOASP framework provides a default implementation for the DSL, e.g. the interpretation of the fact-based language, in several program packages (*.lp files). If advanced features (such as multiple inheritance, automatic symmetry breaking) are required, the default implementation must be replaced with alternative implementations, whereas the OOASP-DSL can stay the same. The OOASP-DSL is largely independent of special ASP features and can therefore be easily translated to other formalisms (OWL/RDF, UML, etc.).

4.1 Defining the knowledge base

The knowledge base comprises an object-model describing types of available components and possible relations between them. In addition, it can include a number of constraints on types and relations. To define the object-model of the configurator with the OOASP-DSL, the following predicates are used where all IDs are considered to be unique within a knowledge base.

```
ooasp_class(KBID,CID)
```

- Defines a class in the knowledge base KBID [3].
  KBID is an id for a knowledge base and CID is an id for a class within the given knowledge base.

```
ooasp_subclass(KBID,CID,SUPERCID)
```

- Defines an inheritance hierarchy of classes. Although other interpretations are possible, in this paper the inheritance hierarchy is assumed to be a tree (single inheritance).

```
ooasp_assoc(KBID,ASSOCID,
            CID1,C1MIN,C1MAX,CID2,C2MIN,C2MAX)
```

- Defines the association between classes CID1 and CID2 within given cardinalities, i.e. for every instance of CID1 there exist at least C2MIN and at most C2MAX instances of CID2 in the association and vice versa.

```
ooasp_attribute(KBID,CID,ATTRID,
                {"string","integer","boolean"})
```

- Defines an attribute for the class CID with the given type.

```
ooasp_attribute_minInclusive(KBID,CID,
                             ATTRID,MINVALUE)
```

- Defines an optional minimum value for integer attributes

```
ooasp_attribute_maxInclusive(KBID,CID,
                             ATTRID,MAXVALUE)
```

- Defines an optional maximum value for integer attributes

```
ooasp_attribute_enum(KBID,CID,
                     ATTRID,ENUMVALUE)
```

- Defines an enum-value (a possible value) for a string attribute.

The mentioned predicates are sufficient to describe the object-model of a simple object-oriented configurator. Many features which can be additionally found in object-oriented systems such as initial values, constants, multi-valued attributes, ordered associations, etc. are currently missing in the framework, but these features are not relevant to the demonstration of the configuration scenarios presented in the paper. In practice, especially ordered associations and initial values are a convenient feature of object oriented product configurators.

Example The OOASP-DSL representation for the modules example corresponds to the following set of facts:

```
% modules example kb "v1"
% classes
ooasp_class("v1","ConfigObject").
ooasp_class("v1","Frame").
ooasp_class("v1","Module").
ooasp_class("v1","ModuleA").
ooasp_class("v1","ModuleB").
ooasp_class("v1","Element").
ooasp_class("v1","ElementA").
ooasp_class("v1","ElementB").

% class inheritance
ooasp_subclass("v1","Frame","ConfigObject").
ooasp_subclass("v1","Module","ConfigObject").
ooasp_subclass("v1","Element","ConfigObject").
ooasp_subclass("v1","ElementA","Element").
ooasp_subclass("v1","ElementB","Element").
```

[2]The ASP code for OOASP is available upon request from the first author

[3]To allow uppercase names, OOASP identifiers are strings, not constants

```
ooasp_subclass("v1","ModuleA","Module").
ooasp_subclass("v1","ModuleB","Module").

% attributes and associations
% class Frame
ooasp_assoc("v1","Frame_modules",
  "Frame",1,1,
  "Module",0,5).

% class Module
ooasp_attribute("v1","Module","position","integer").
ooasp_attribute_minInclusive("v1",
  "Module","position",1).
ooasp_attribute_maxInclusive("v1",
  "Module","position",5).

% class Element
ooasp_assoc("v1","Element_module",
  "Element",1,1,
  "Module",1,1).
```

4.2 Defining a configuration

A (partial) configuration is an instantiation of the object-model. A valid configuration is a configuration where no constraint is violated. It represents a buildable artifact of the configured system.

As with knowledge-bases, OOASP allows the representations of multiple configurations within one ASP program. We use the following predicates to define a (partial) configuration:

`ooasp_configuration(KBID, CONFIGID)`

- Declares that the configuration CONFIGID belongs to the knowledge base KBID. Every configuration has a unique ID and belongs to exactly one knowledge base.

`ooasp_isa(CONFIGID, CID, OBJECTID)`

- The object with the id OBJECTID is an instance of the class CID in the configuration CONFIGID. If an object is an instance of a class, it must also be an instance of one of its leaf classes (i.e. class without subclasses).

```
ooasp_associated(CONFIGID,ASSOCID,
                 OBJECTID1,OBJECTID2)
```

- The objects with the OBJECTID1 and OBJECTID2 are associated in the association ASSOCID in the configuration CONFIGID.

```
ooasp_attribute_value(CONFIGID,ATTRID,
                      OBJECTID,VALUE)
```

- The attribute ATTRID of the object OBJECTID has the value VALUE in the configuration CONFIGID.

Example The following configuration consisting of one frame, one module, and one element is not valid. It would be valid if the module 10 was a ModuleB.

```
ooasp_configuration("v1","c1").
ooasp_isa("c1","Frame",1).
ooasp_isa("c1","ModuleA",10).
ooasp_attribute_value("c1","position",10,5).
ooasp_isa("c1","ElementB",20).
ooasp_associated("c1","Frame_modules",1,10).
ooasp_associated("c1","Element_module",20,10).
```

4.3 Defining constraints

There are two different kinds of constraints in OOASP: integrity constraints and domain-specific constraints. Both are implemented as ASP rules which derive an atom `ooasp_cv` (head) for each constraint violation expressed in the body. The derived atom can be used for explanations.

Integrity constraints are generic constraints derived from the object model of the knowledge base. Implementations of integrity constraints are provided by the OOASP framework in program package ooasp_check.lp, e.g. for the constraint which checks the minimal cardinality of associations:

```
% Derive ooasp_cv(CONF,mincardviolated(ID1,ASSOC))
% whenever there are less objects associated
% with object ID1 than allowed by the cardinality
% restriction of the association
ooasp_cv(CONF,mincardviolated(ID1,ASSOC)) :-
  { ooasp_associated(CONF,ASSOC,ID1,ID2):
    ooasp_isa(CONF,C2,ID2) } C2MIN-1,
  C2MIN>0,
  ooasp_isa(CONF,C1,ID1),
  ooasp_assoc(KBID,ASSOC,
            C1,C1MIN,C1MAX,C2,C2MIN,C2MAX),
  ooasp_configuration(KBID,CONF).
```

In addition to the integrity constraints, a knowledge engineer can define domain-specific constraints for a knowledge base. These are constraints that can not be derived automatically from the knowledge base.

Example The first of the constraints in the modules examples may be implemented as follows:

```
% ElementA requires ModuleA
ooasp_cv(CONF,wrongModuleType(E,M)) :-
  ooasp_configuration("v1",CONF),
  ooasp_associated(CONF,"Element_module",E,M),
  ooasp_isa(CONF,"ElementA",E),
  not ooasp_isa(CONF,"ModuleA",M).
```

5 Product Configuration Scenarios

This section describes some of the typical scenarios for an object-oriented product configurator.

5.1 Checking a Configuration

Checking a (partial) configuration evaluates the integrity constraints of the knowledge base and the domain-specific constraints for a configuration under closed world assumption, i.e. during the checking no new objects are instantiated.

In an interactive configurator, checking the current configuration highlights the parts of the configuration that need to be changed by a user.

Example Checking the minimal configuration consisting of only one element of type A

`ooasp_isa("c2","ElementA",10).`

will derive a cardinality violation

`ooasp_cv("c2",mincardviolated(10,"Element_module")).`

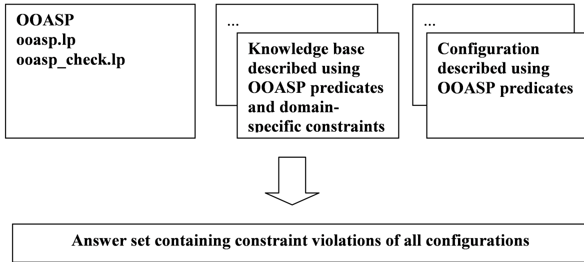

Figure 2: Checking a configuration

for the association between element and module, indicating that there must be a module for the element with OBJECTID 10.

The constraint violations derived during checking can also guide a repair-based solver to repair the current configuration [Falkner *et al.*, 2011]. If checking does not find any constraint violation, the current configuration is valid. The process of checking a configuration within OOASP framework is depicted in Figure 2.

5.2 Completing a Configuration

Given a possible empty (partial) configuration, find an extension of the configuration that satisfies all constraints. No fact of the given configuration may be removed. If no such extension can be found, the current configuration is either inconsistent or there is no valid configuration with the given upper bounds for object instances.

Completing a configuration can be accomplished by enumerating all possible extensions of the given configuration until a valid configuration is found. Figure 3 shows the necessary program packages for completing a configuration.

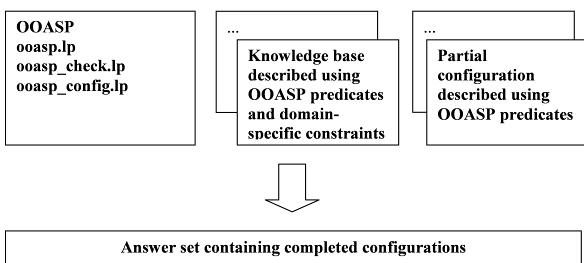

Figure 3: Completing a configuration

To enumerate all possible configurations one has to instantiate objects according to customer requirements. The number of the possible instances is controlled by the predicate

```
ooasp_domain(CONFIGID, CID, OBJID)
```

- The object with the OBJID can be instantiated to one of the leaf-classes of class CID.

The `ooasp_domain` facts define the available object IDs for a configuration. The object IDs are unique within a configuration. Every object ID can represent one instance of a leaf-class. However, the classes used in the `ooasp_domain` predicates can be non-leaf-classes as well. Therefore, the number of `ooasp_domain` facts for each class CID is equal to the maximal number of its instances in the configuration CONFIGID. From the `ooasp_domain` facts the possible types of every object ID in a configuration are derived (`ooasp_canbe`), searching up and down the class hierarchy (see (1) in Figure 4). The `ooasp_isa` facts (2) are derived upwards only. This approach of controlling instantiation is similar to the notion of a scope in Alloy [Jackson, 2011].

Example `ooasp_domain("c3","Module",20)` allows the object with OBJECTID 20 to become either a ModuleA or a ModuleB but not Frame. In a second step it may be set explicitly to be a ModuleA. Figure 4 shows the derived information after the object has been instantiated this way.

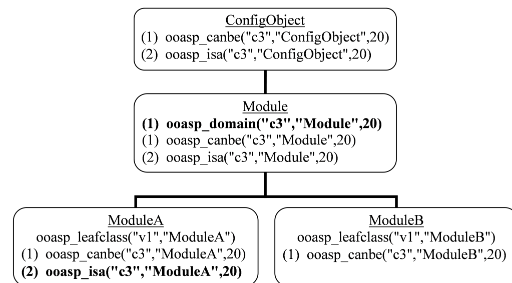

Figure 4: Controlling instantiation

The enumeration of all possible configurations is accomplished by instantiating objects and setting associations and attributes. The default implementation for instantiating objects is done in program package ooasp_config.lp:

```
% instantiate objects
0 { ooasp_isa(CONF,LEAFCLASS,ID) :
    ooasp_leafclass(V,LEAFCLASS) :
    ooasp_canbe(CONF,LEAFCLASS,ID) } 1 :-
  ooasp_domain(CONF,C,ID),
  ooasp_configuration(V,CONF).
```

This means that every object ID can become an instance of one of its possible leaf-class types. Associations are set in a similar matter. Every instance in the configuration can be associated with all other possible instances in the configuration. Constraints ensure that only instantiated objects are associated.

```
% associate objects
C2MIN { ooasp_associated(CONF,ASSOC,ID1,ID2):
        ooasp_canbe(CONF,C2,ID2) } :-
  ooasp_isa(CONF,C1,ID1),
  ooasp_assoc(V,ASSOC,C1,C1MIN,C1MAX,C2,C2MIN,C2MAX),
  ooasp_configuration(V,CONF).
```

```
% type check - only use instantiated objects
:- ooasp_associated(CONFIG,ASSOC,ID1,ID2),
   not ooasp_isa(CONFIG,C2,ID2),
   ooasp_assoc(V,ASSOC,C1,C1MIN,C1MAX,C2,C2MIN,C2MAX),
   ooasp_configuration(V,CONFIG).
```

Finally, there must be a generating rule for all possible values of the attributes of an object. The following shows the generating rule for integer attributes.

```
% set attribute values for integer attributes
1 { ooasp_attribute_value(CONFIG,N,ID,VALUE):
    VALUE=MIN..MAX } 1 :-
  ooasp_attribute(V,C,N,T),
  ooasp_isa(CONFIG,C,ID),
  ooasp_attribute_minInclusive(V,C,N,MIN),
  ooasp_attribute_maxInclusive(V,C,N,MAX),
  ooasp_configuration(V,CONFIG).
```

Example Figure 5 shows a completed configuration for the partial configuration c3 below. It contains three instances of ElementA and two instances of ElementB. Note that `ooasp_isa` is given as customer requirement only for those elements. For modules, only the `ooasp_domain` is given and only 5 of the available 10 IDs are used in the completed configuration.

```
% Partial configuration
ooasp_configuration("v1","c3").
ooasp_domain("c3","Frame",1).
ooasp_domain("c3","ElementA",10..12).
ooasp_isa("c3","ElementA",10..12).
ooasp_domain("c3","ElementB",13..14).
ooasp_isa("c3","ElementB",13..14).
ooasp_domain("c3","Module",20..29).
```

Frame 1

Module A20	Module A21	Module A22	Module B23	Module B24
Element A10	Element A11	Element A12	Element B13	Element B14

Figure 5: Complete configuration for the modules example

5.3 Reconciliation

Given a complete legacy configuration and the changed knowledge base which makes the configuration invalid, find a new valid configuration that is close to the legacy configuration.

Reconciliation of a configuration is illustrated in Figure 6. OOASP uses the same cost-based reconciliation approach as described in [Friedrich *et al.*, 2011]. For every change in the legacy configuration, a cost can be defined. This allows a fine control over the reconfiguration process. The optimal reconciliation is the reconfiguration that minimizes the costs. For example, the rule for reconciling associations either keeps the link between two objects in the legacy configuration or removes it. Reconciliation is controlled by the following predicates:

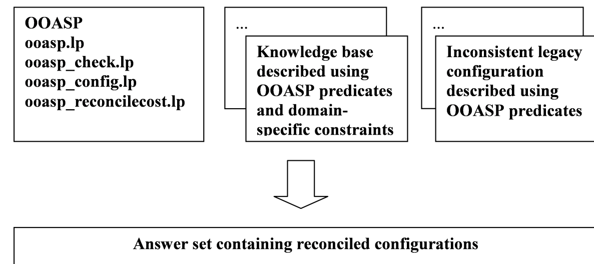

Figure 6: Reconcile a configuration

`ooasp_reconcile(LEGACY,RECONCILED)`

- Activates reconciliation from configuration LEGACY to the configuration RECONCILED

`ooasp_cost_instance(KB,CID,ADD,REMOVE)`

- Defines the costs for adding and removing instances of class CID

`ooasp_cost_assoc(KB,ASSOC,ADD,REMOVE)`

- Defines the costs for adding and removing a link to/from the association ASSOC

`ooasp_cost_attribute(KB,ATTR,COST)`

- Defines the cost for changing attribute ATTR

`ooasp_rcost(CHANGEOFLEGACYCONFIGURATION,COST)`

- For every modification of the legacy configuration an `ooasp_rcost` atom is derived, defining the COST of the modification. The best reconciliation is the one that minimizes the overall cost of the `ooasp_rcost` atoms, i.e. `#minimize[ooasp_rcost(CHANGE,COST)=COST]`.

The following listing shows the implementation of the rules for reconciling associations:

```
% either reuse link or remove it:
% ooasp_remove_associated is derived,
% if a link is removed
1 { ooasp_associated(RECONCILED,ASSOC,ID1,ID2),
    ooasp_remove_associated(RECONCILED,ASSOC,
                            ID1,ID2) } 1 :-
  ooasp_associated(LEGACY,ASSOC,ID1,ID2),
  ooasp_reconcile(LEGACY,RECONCILED).

% derive the reconfiguration costs
% ooasp_rcost contains the overall costs
ooasp_rcost(ooasp_remove_assoc(ID1,ID2),REMOVE) :-
  ooasp_remove_associated(RECONCILED,ASSOC,ID1,ID2),
  ooasp_cost_assoc(KB,ASSOC,ADD,REMOVE),
  ooasp_configuration(KB,RECONCILED),
  ooasp_reconcile(LEGACY,RECONCILED).
```

Example Suppose after the first systems of our example domain have been built, there is evidence of a previously unknown overheating problem if two modules of type A are put next to each other in a frame. Thus, to prevent overheating we have to add a new constraint to the knowledge base that disallows putting two modules of type A next to each other.

```
%  do not put 2 modules of type moduleA
%  next to each other
ooasp_cv(CONF,moduleANextToOther(M1,M2,P1,P2)):-
  ooasp_configuration("v2",CONF),
  ooasp_associated(CONF,"Frame_modules",F,M1),
  ooasp_associated(CONF,"Frame_modules",F,M2),
  ooasp_attribute_value(CONF,"position",M1,P1),
  ooasp_attribute_value(CONF,"position",M2,P2),
  M1!=M2,
  ooasp_isa(CONF,"ModuleA",M1),
  ooasp_isa(CONF,"ModuleA",M2),
  P2=P1+1.
```

Frame 1

Module A20	Module B24	Module A22	Module B23	Module A21
Element A10	Element B14	Element A12	Element B13	Element A11

Figure 7: Reconciled configuration for the modules example

Because of the added constraint the legacy configuration in Figure 5 is no longer valid. Using reconciliation with equal costs for all changes to the configuration results in the new configuration shown in Figure 7.

5.4 Choosing the best knowledge base for reconciliation

Given a new technical requirement and N knowledge bases satisfying that requirement, choose the knowledge base that minimizes the costs for reconciling legacy configurations and the estimated costs for building a new system and maintaining existing systems.

Note that the costs for maintaining systems may also contain the costs for future reconciliations. Often there are many different technical solutions satisfying new requirements affecting existing systems. The choice of a technical solution that minimizes the costs for reconciliation of the legacy systems is an important problem to be solved. Given costs for various system modifications, we have to find a solution which corresponds to the most cost-effective reconciliation.

Example A possible technical solution for the overheating modules is to avoid putting the modules next to each other. Suppose there is an alternative technical solution replacing module A with a new module ANEW, which does not have the overheating problem.

Frame 1

Module ANEW 30	Module ANEW 31	Module ANEW 32	Module B23	Module B24
Element A10	Element A11	Element A12	Element B13	Element B14

Figure 8: Reconciled configuration with the module of type ANEW

Reconcilation in the alternative knowledge base consists in replacing modules of type A with modules of type ANEW, but no rearranging is necessary. The result is shown in Figure 8. If modules of type A can be used together with type ANEW then it is sufficient to just replace module A 21 with module ANEW 31.

Which technical solution shall be chosen? To answer this question one has to find the affected configurations. With a framework like OOASP, the effected legacy configurations (i.e. deployed systems which must be reconfigured) can be computed by checking the constraint representing the new technical requirement in all available legacy configurations. Note that legacy configurations may use earlier versions of the knowledge base. In this case the legacy configurations must be upgraded to the current version of the knowledge base or the constraint must be expressed in terms of the legacy knowledge bases.

Using the reconcile scenario one can compute how costly it would be to modify the existing legacy configurations to the available technical solutions.

The cost for new systems can be estimated by computing the configuration cost of existing legacy systems, i.e. how costly it would have been to build these systems from scratch with the new knowledge bases. This can be computed by a configurator using the initial (partial) configuration, i.e. the customer requirements and completing the configuration with the new knowledge base.

The costs for future reconciliations are hard to compute in general, unless there is some knowledge about the future requirements. Otherwise, one has to estimate how often the critical constellations will occur. By concentrating on the most probable reconcile scenarios of a product configurator, one can simulate these reconciliation scenarios using alternative knowledge bases and compare their costs.

6 Evaluation

The main purpose of OOASP is to demonstrate the behavior of an object-oriented configurator within a logical framework. Therefore, performance was not the main focus of this paper. Since OOASP uses a similar approach as [Friedrich *et al.*, 2011], its performance is similar, too.

For checking configurations, the framework proved to be able to handle more than 1000 components for integrity con-

straints and simple domain-specific constraints. Of course, one can always come up with complex constraints, like finding all paths in a graph, for which computation of logical models is infeasible.

Completing a configuration can be handled for problems with hundreds of components. The main limiting factor here is the grounding size. The grounding explodes since the generic translation of UML to ASP rules of the default OOASP implementation associates every possible object of one related type with every other possible object of the other type. The grounding of the module example with 200 objects is already greater than 500 MB. One can reduce the grounding size by replacing the rules of the generic translation with special instantiation rules as follows:

```
% associate objects
% special implementation
% 'create' a module for every element
% at a fixed object ID
1 {ooasp_associated(CONF,
    "Element_module",
    ID1,1000+ID1)} 1:-
  ooasp_isa(CONF,"Element",ID1),
  ooasp_configuration(V,CONF).
```

This is similar to automatically generating subobjects in an object-oriented setting. However, these special rules can no longer be used for reconfiguration, because they assume that for every element there is a unique module at a fixed object ID. Another way to avoid the explosion of grounding size would be to use a constraint-based model. Additional limiting factor is the current lack of ASP to incorporate domain-specific heuristics into the solving, which is a topic of active research.

7 Conclusions

This paper demonstrates the implementation of a small object-oriented product configurator on top of ASP. The framework contains a domain-specific language for specifying knowledge bases and configurations, that can be easily translated to other formalisms (OWL/RDF, UML/Java).

Evaluations showed that checking constraints relative to a given configuration can be done effectively. However, finding (re)configurations efficiently remains a challenge for large-scale product configuration. The main obstacle for SAT- and ASP-based approaches seems to be the explosion of grounding. In addition, the identification of appropriate domain-specific heuristics is an open problem for all search-based approaches.

By defining typical configuration scenarios we hope to raise awareness to often neglected aspects of product configuration. We demonstrated the handling of these scenarios in ASP and are going to continue this work for other formalisms such as constraint programming, RDF/OWL, etc.

References

[Brewka *et al.*, 2011] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[Eiter *et al.*, 2009] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web*, pages 40–110, 2009.

[Falkner and Haselböck, 2013] Andreas Falkner and Alois Haselböck. Challenges of Knowledge Evolution in Practice. *AI Communications*, 26:3–14, 2013.

[Falkner *et al.*, 2011] Andreas Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner. Modeling and solving technical product configuration problems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 25:115–129, 2011.

[Felfernig *et al.*, 2004] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213–234, 2004.

[Friedrich *et al.*, 2011] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner. (Re)configuration based on model generation. In Conrad Drescher, Ins Lynce, and Ralf Treinen, editors, *LoCoCo*, volume 65 of *EPTCS*, pages 26–35, 2011.

[Gebser *et al.*, 2011] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *5th International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[Jackson, 2011] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. Mit Press, 2011.

[Rumbaugh *et al.*, 2005] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2 edition, 2005.

[Soininen and Niemel, 1998] Timo Soininen and Ilkka Niemel. Formalizing configuration knowledge using rules with choices. In *Seventh International Workshop On Nonmonotonic Reasoning*, 1998.

[Soininen *et al.*, 2001] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge*, pages 195–201, 2001.