OOASP: Connecting Object-oriented and Logic Programming*

Andreas Falkner[1], Anna Ryabokon[2], Gottfried Schenner[1], and Kostyantyn Shchekotykhin[2]

[1] Siemens AG Österreich, Vienna, Austria,
andreas.a.falkner@siemens.com gottfried.schenner@siemens.com
[2] Alpen-Adria-Universität Klagenfurt, Austria,
anna.ryabokon@aau.at kostyantyn.shchekotykhin@aau.at

Abstract

Most of contemporary software systems are implemented using an object-oriented approach. Modeling phases – during which software engineers specify requirements to the future system using some modeling language – are an important part of the development process, since modeling errors are often hard to recognize and correct. OOASP framework tries to solve the problem by embedding Answer Set Programming into the object-oriented software development process. Preliminary results of the OOASP application in CSL Studio, which is an internal modeling environment of Siemens for product configurators, show that it can be used as a lightweight approach to verify, create and transform instantiations of object models at runtime and to support the software development process during design and testing.

1 System overview

Object-oriented programming languages are de facto standard for software development. In practice of Siemens the object-oriented approach is used in many domains among which development of product configurators is one of the prominent examples. A configurator is a software system that enables the design of 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 functional and technical properties of real-world and abstract components of a configurable product. An attribute takes values from

*This research was funded by the Austrian Research Promotion Agency (grant number 840242), Carinthian Science Fund (grant number KWF-3520/26767/38701) and Austrian Science Fund (grant number I 2144 N-15)

a predefined domain. Furthermore, components can be related/connected to each other in various ways.

Siemens experience in the development of industrial configurator applications shows that quite often incorrect models are responsible for faults in software artifacts [6]. It is difficult to get models right in the first place due to unexpected effects of development decisions. To solve this problem we suggest the OOASP approach which allows to analyze object-oriented software models and their instances by means of ASP. In particular, we consider models which can be described by a modeling language corresponding to a UML class diagram [13]. The latter is a language allowing a software developer to specify an object model and additional constraints that each valid instantiation of an object model must satisfy. OOASP was implemented as a potential extension to any object-oriented modeling environment and its practicability was evaluated together with CSL Studio [3] which is a Siemens internal tool for the design of product configurators based on the methodology of Generative Constraint Satisfaction Problems (GCSPs) [7, 17]. CSL (Configuration Specification Language) is a formal modeling language based on a standard object-oriented meta-model similar to Ecore[1] or MOF[2]. It provides all state-of-the-art features such as packages, interfaces, enumerations, classes with attributes of various types, associations between classes, inheritance and aggregation relations as well as integrity constraints for subtypes and cardinalities of associations. In addition, it offers reasoning methods such as rules and constraints which are not (yet) automatically mapped to OOASP and therefore are not covered in this work.

In OOASP, all concepts of one or multiple software models as well as their instantiations are represented in terms of the Domain Description Language (DDL). An OOASP-DDL program comprises facts encoding the object-oriented classes, attributes, associations and integrity constraints (see Section 2). In order to reason about a software model specified in DDL, OOASP framework uses a *meta-programming* approach [16] which was successfully applied in a similar way, for instance, to debugging of ASP programs [9, 12]. In case of OOASP a meta-program corresponds to a set of normal rules and/or their extensions, such as choice rules (see e.g. [1, 4, 10, 14] for an introduction to ASP). In a standard OOASP implementation we provide meta-programs accomplishing the following tasks[3]:

Validation Given an OOASP-DDL program describing an object-oriented model and its instantiation, a validation meta-program verifies whether all integrity and domain-specific constraints hold. The integrity constraints encode model requirements to relations between objects of an instantiation and are derived from the given model automatically. The domain-specific constraints ensure that some specific requirements to an instantiation of a model are satisfied. They can either be directly specified in the meta-program or imported from other languages. For instance, one could import domain-specific constraints

[1] Eclipse Modeling Framework https://www.eclipse.org/modeling/emf/

[2] MetaObject Facility http://www.omg.org/mof/

[3] OOASP code and encodings are available upon request from the first author.

defined in Object Constraint Language[4] (OCL), for which transformations to SAT [15] and constraints programming [2] exist.

Completion Given an OOASP-DDL program describing an object-oriented model and its (partial) instantiation (where an empty instantiation can be seen as a special case) the completion task is to find an extension of the instantiation that satisfies all constraints or to show that such extension does not exist. The latter may occur due to two main reasons: (i) the object-oriented model or the given (partial) instantiation are inconsistent and do not have a completion; and (ii) the extension of the given instantiation requires the creation of a number of objects that exceeds a given upper bounds for object instances. This bounding is necessary for reducing the search space to a manageable size and relies on the assumption that a solution using a sufficiently small number of constants exists - similar to the notion of a scope in Alloy [11].

Reconciliation Given an OOASP-DDL program describing a legacy instantiation of an outdated object-oriented model, a new up-to-date model and a set of transformation rules, the goal of the reconciliation is to find a possibly preferred set of changes required to transform the legacy instantiation to a valid instantiation of the new model. The preferences in OOASP can be defined with domain-specific costs that assess the costs of required changes such as creation, reuse or disposal (deletion) of object instances.

If advanced features such as multiple inheritance, symmetry breaking, etc., are required, the default ASP encodings of reasoning tasks, outlined in this paper, must be replaced with alternative encodings, whereas the OOASP-DDL program remains the same.

A typical workflow of the product configurator development process in CSL Studio and OOASP is depicted in Figure 1. The development starts with the creation of an initial configuration model and the definition of domain-specific constraints in CSL. Model and constraints are then transformed into JCOS format and are embedded into the configuration application. The latter implements the Graphical User Interface (GUI) and provides reasoning services by means of a generative constraint solver JCOS [17, 7]. On every stage of the configurator development process the application is tested for correctness. The testing process starts with the definition of instances in CSL Studio. After importing them into the JCOS solver, a tester uses the GUI to check functionality of the application. The results of these tests can be exported from the application back to CSL Studio and be used to extend the existing test instances. In addition, the model and test instances can be exported to OOASP. The export procedure generates all necessary DDL definitions automatically. The domain-specific constraints of CSL, however, have to be reimplemented by the developer in ASP. This redundancy allows the framework to increase the probability of finding faults in domain-specific constraints, which are among the most error-prone parts of a configuration application.

[4] OCL specification is available from http://www.omg.org/spec/OCL/2.4/PDF/

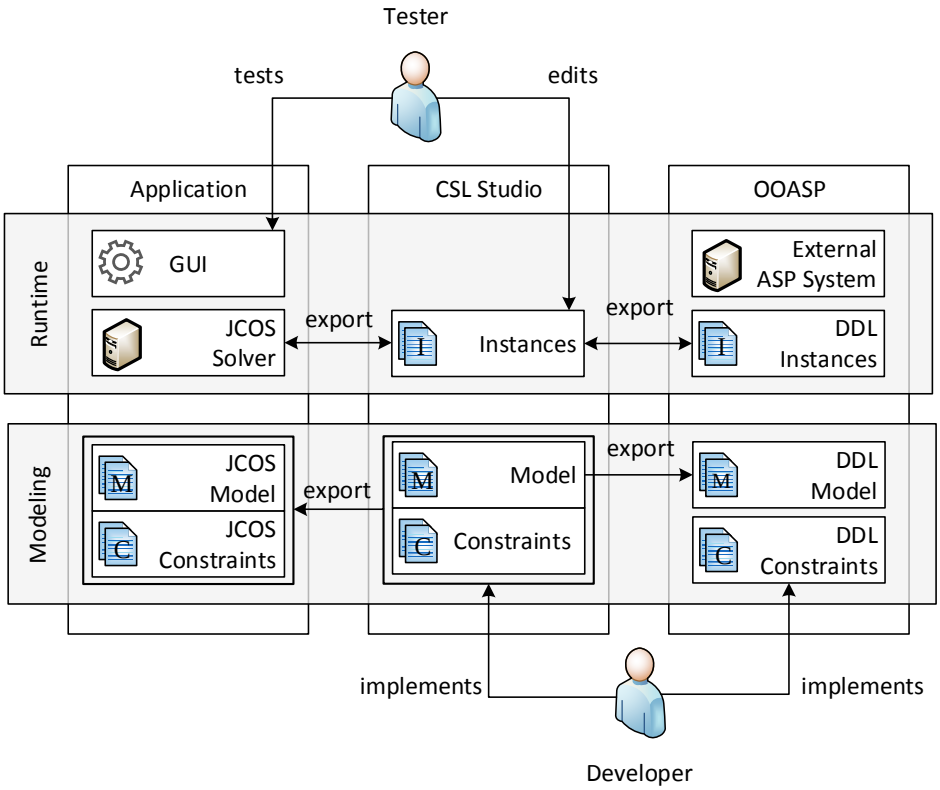

Figure 1: Integration of OOASP in the development of product configurators

In general, OOASP can be used together with any UML/OCL based framework. The ”CSL Studio” in Figure 1 would be replaced with another modeling environment and/or the ”Application” with another runtime environment and solver. For verification and test of the model including the constraint implementation, we see the following approaches (see Section 4 for more details and examples):

- Verify the consistency of the whole model (existence of at least one instance): Execute the completion task with an empty instantiation.
- Verify the correctness of domain-specific constraints: Define appropriate (partial) instantiations and compare the results of application and OOASP. If the OOASP constraints are automatically translated from the model (i.e. no diverse redundance) then the expected truth value must additionally be given.
- Verify the checking capabilities of the application: In the application, create positive and negative test instances (product configurations) and execute OOASP validation task to cross-check whether the positive configurations are valid instantiations of the model, while the negative ones are not.

- Verify the generation capabilities of the application, e.g. adding missing components and connecting them to components existing in the partial configuration: Define partial instantiations, execute both application and OOASP completion task on them and compare the results. Typically, the application will have implemented some heuristics (necessary for solving large problem instances) which can thus be tested on examples sufficiently small for OOASP complete solving.

- Verify the repair capabilities of the application: If the software developer (tester) manipulates a completed configuration, for instance, by adding or removing components, OOASP can help to restore consistency through the reconciliation task. It finds a set of changes that keeps as much of the existing structure of the configured system as possible.

2 OOASP Domain Description Language

OOASP-DDL allows a software developer to define all standard concepts of object-oriented models such as classes, attributes and associations. Each concept of the model is translated to a corresponding OOASP-DDL atom, where each term Id_* is an identifier of a model, class, attribute, etc. In OOASP identifiers of models are globally unique, whereas all other identifiers are unique within a model. In the current version, OOASP-DDL supports the definitions presented in Table 1. These definitions are sufficient to describe a subset of the object-oriented model of programming languages such as C++, Java, etc. Many features that can additionally be found in object-oriented models, e.g. initial values, constants, multi-valued attributes, ordered associations, etc., are currently not supported by the framework. This is because our main purpose was to provide a lightweight approach that, however, is able to capture most of the features commonly used in practice. The definition of an instantiation of an object-oriented model is done using OOASP-DDL in a similar way as the definition of the model. In particular, our language allows the definitions shown in Table 2.

Note that, OOASP-DDL is designed in a way to allow the definition of multiple models and their instantiation in one ASP program. This provides the necessary support for reconciliation and similar reasoning tasks that are applied to many models and/or their instantiations at once.

3 Definition of constraints

Constraints allow a software developer to ensure that models and their instantiations are valid. In OOASP we support two types of constraints: integrity constraints and domain-specific constraints. The latter are used to verify some specific properties of a model and/or its instantiations. The definition of domain-specific

$ooasp\_class(Id_M, Id_C)$	defines a class C in a model M
$ooasp\_subclass(Id_M, Id_C, Id_{SC})$	defines a subclass relation between a class C and a super class SC in a model M
$ooasp\_assoc(Id_M, Id_A, Id_{C_1}, Min_{C_1}, Max_{C_1}, Id_{C_2}, Min_{C_2}, Max_{C_2})$	defines an association relation A between classes C_1 and C_2 with the given cardinalities, e.g. for every instance of the class C_1 at least Min_{C_2} and at most Max_{C_2} instances of the class C_2 must be associated
$ooasp\_attribute(Id_M, Id_C, Id_{AT}, \{$*"string","integer","boolean"*$\})$	defines an attribute AT of a class C with one of the three possible types
$ooasp\_attribute\_minInclusive(Id_M, Id_C, Id_{AT}, MinV)$	provides an optional minimum value $MinV$ for an integer attribute AT
$ooasp\_attribute\_maxInclusive(Id_M, Id_C, Id_{AT}, MaxV)$	provides an optional maximum $MaxV$ for an integer attribute AT
$ooasp\_attribute\_enum(Id_M, Id_C, Id_{AT}, Val)$	defines a possible value Val for a string attribute AT

Table 1: OOASP-DDL definitions for the encoding of models

constraints can be done by a developer directly in OOASP-DDL or – if that transformation is available – by importing them from the input model, e.g. OCL constraints from a UML model. The integrity constraints, however, are included in the default OOASP implementation and capture the requirements of the input object-oriented model such as cardinality restrictions, typing, etc. For instance, in order to ensure that a minimal cardinality requirement of an association relation holds in a given instantiation, OOASP framework comprises the following rule[5]:

```
1 ooasp_cv(I,mincardviolated(O1,A)) :-
2    {ooasp_associated(I,A,O1,O2): ooasp_isa(I,C2,O2)} C2MIN-1,
3    C2MIN>0,
4    ooasp_assoc(M,A,C1,C1MIN,C1MAX,C2,C2MIN,C2MAX),
5    ooasp_instantiation(M,I),
6    ooasp_isa(I,C1,O1).
```

The presence of an atom over *ooasp_cv* predicate in an answer set of an OOASP program indicates that the corresponding integrity constraint is violated by the given instantiation. In the sample rule above, the error atom is derived whenever less objects of type C_2 are associated with object O_1 than required by the

[5]In our examples we use the gringo [8] dialect of ASP that also allows usage of uninterpreted function symbols such as *mincardviolated*.

$ooasp\_instantiation(Id_M, Id_I)$	defines an instantiation I of a model M
$ooasp\_isa(Id_I, Id_C, Id_O)$	declares that an object O is an instance of the class C in instantiation I
$ooasp\_associated(Id_I, Id_A, Id_{O_1}, Id_{O_2})$	connects objects O_1 and O_2 via the association relation A
$ooasp\_attribute\_value(Id_I, Id_{AT}, Id_O, Val)$	assigns a value Val to an attribute AT of an object O

Table 2: OOASP-DDL definitions for the encoding of instantiations

cardinality restriction of the association.

4 OOASP tasks

In this section we consider OOASP tasks listed in Section 1 in more detail. Assume a developer programs a simple hardware configuration problem shown in Figure 2. The sample model describes a product configuration problem as a UML class diagram. In this problem the hardware product consists of a number of *Frames*. Each frame contains up to five *Modules* of types *ModuleA* or *ModuleB*, where each module occupies exactly one of the 5 positions in a frame. Moreover, each module has exactly one *Element* assigned to it. All elements are of one of two types *ElementA* or *ElementB*. The corresponding OOASP-DDL encoding for this example is automatically generated by CSL Studio. A part of the encoding excluding integrity constraints is shown in Listing 1.

Additionally to the integrity constraints, implied by the cardinalities of associations shown on the UML diagram, there are the following domain-specific constraints:

- Elements of type *ElementA* require a module of type *ModuleA*
- Elements of type *ElementB* require a module of type *ModuleB*
- Modules must occupy different positions in a frame

These constraints can easily be implemented in OOASP. For instance, the first and the third can be formulated as shown in Listing 2.

4.1 Validation of a configuration

The implementation of an object-oriented software requires continuous testing in order to identify and resolve faults early. The validation reasoning task provided by OOASP allows a software developer to verify whether an instantiation generated by the object-oriented code is consistent. Especially, the validation is important in the context of CSL Studio or similar systems while testing domain-specific constraints.

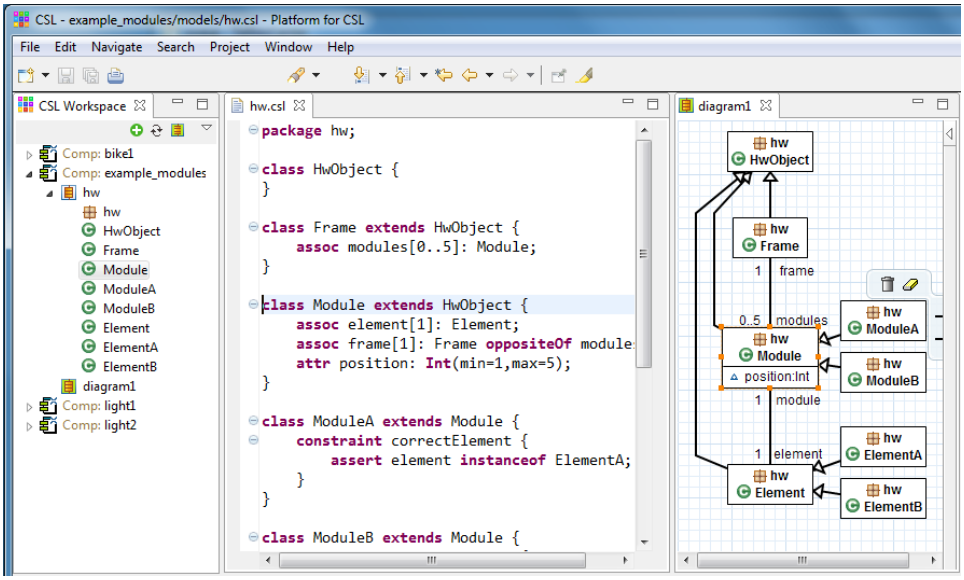

Figure 2: CSL screenshot for the Modules example

In CSL Studio an instantiation of the object model provided by the developer is exported to OOASP and the validation meta-program is executed. The resulting answer set shows the violated requirements which inhibit a valid configuration. Thus, the developer can identify the faults in the software in a shorter period of time.

For instance, assume a software developer designed a model in CSL Studio and the corresponding JCOS application outputs an instantiation `c1` comprising only one element of type `ElementA`. CSL Studio forwards this instantiation to OOASP which translates it to the OOASP-DDL program:

```
ooasp_isa("c1","ElementA",10).
```

For this input, execution of the validation task returns an answer set comprising:

```
ooasp_cv("c1",mincardviolated(10,"Element_module"))
```

This atom indicates that cardinality restrictions of the association between Element and Module classes are violated. The reason is that for the object with identifier `10` there is no corresponding object of the Module type.

Note that in the current OOASP prototype domain-specific constraints must be coded by a software developer manually and are not generated from the CSL constraint language. However, this behavior was found to be advantageous in practice, since it provides a mechanism for the *diverse redundance* [5]. The latter refers to the engineering principle that suggests application of two or more systems. These systems are built using different algorithms, design methodology, etc., to perform the same task. The main benefit of the diverse redundance is that it allows software developers to find hidden faults caused by design flaws which are usually

```
% modules example kb "v1"
% classes
ooasp_class("v1","HwObject").
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","HwObject").
ooasp_subclass("v1","Module","HwObject").
ooasp_subclass("v1","Element","HwObject").
ooasp_subclass("v1","ElementA","Element").
ooasp_subclass("v1","ElementB","Element").
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).
```

Listing 1: OOASP-DDL encoding of the Modules example shown in Figure 2

hard to detect. Generally, we found that software developers are able to formulate domain-specific constraints in OOASP after a short training. However, existence of ASP development environments supporting debugging and testing of ASP programs would greatly simplify this process.

4.2 Completion of an instantiation

The completion task is often applied in situations when a software developer needs to generate a test case for a production system that outputs an invalid instantiation. Thus, the completion task allows a developer to detect two types of problems: (i) invalid partial instantiation and (ii) incomplete partial instantiation. In the last case, the partial instantiation returned by a configurator can be extended to a valid one by adding missing objects and/or relations between them. This indicates that the

```
ooasp_cv(I,module_element_violated(M1,E1)) :-
    ooasp_instantiation(M,I),
    ooasp_associated(I,"Element_module",M1,E1),
    ooasp_isa(I,"ElementA",E1),
    not ooasp_isa(I,"ModuleA",M1).

ooasp_cv(I,alldiffviolated(M1,M2,F)) :-
    ooasp_instantiation(M,I),
    ooasp_isa(I,"Module",M1),
    ooasp_isa(I,"Module",M2),
    ooasp_attribute_value(I,"position",M1,P),
    ooasp_attribute_value(I,"position",M2,P),
    ooasp_associated(I,"Frame_modules",F,M1),
    ooasp_associated(I,"Frame_modules",F,M2),
    M1 != M2.
```

Listing 2: Sample domain-specific constraints in OOASP

already implemented production system works correctly, at least for the given input, but it is incomplete. The developer can export the obtained solution and use it as a test case during subsequent implementation of the system. If the problem of the first type is found, then we have to differentiate between two causes of this problem: (a) the model designed in CSL Studio is inconsistent; and (b) the system returned a partial instantiation that is faulty, i.e. cannot be extended to a valid solution. The first cause can easily be detected by running a completion task with an empty instantiation. If the model is consistent, then manually coded additional constraints of the production system are faulty and the software developer has to correct them.

In order to execute the completion task the CSL Studio exports an instantiation obtained by an object-oriented system to OOASP-DDL. Then, this instantiation together with the completion meta-program is provided to an ASP solver. The returned answer sets are visualized by the system to the software developer. If needed, the developer can export the found complete instantiation to an instantiation of the object-oriented system. This translation is straight-forward due to the one-to-one correspondence between instances on the OOASP-level and the object-oriented system. The completed configuration can either be used for further tests in the object-oriented application or just be compared to the result of the same completion action in an automated test.

Consider the following example in which a partially implemented configuration system returns an instantiation containing three instances of `ElementA` and two instances of `ElementB`.

```
% Partial configuration
ooasp_instantiation("v1","c2").
```

```
3 ooasp_isa("c2","ElementA",10). ooasp_isa("c2","ElementA",11).
4 ooasp_isa("c2","ElementA",12).
5 ooasp_isa("c2","ElementB",13). ooasp_isa("c2","ElementB",14).
```

In this case the completion task returns a solution visualized in Figure 3. This solution comprises the existing objects with identifiers $10-14$ as well as the new objects corresponding to a frame with object identifier 30 and five modules $20-24$.

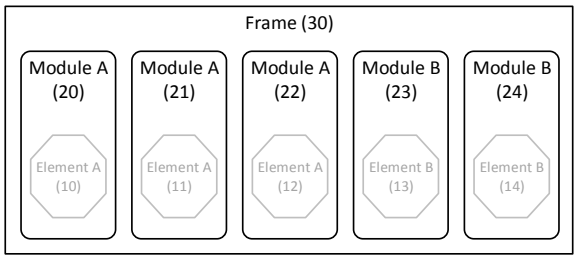

Figure 3: Complete instantiation for the Modules example. The objects existing in the input instantiation are shown in gray.

4.3 Reconciliation of an inconsistent instantiation

The reconciliation task deals with restoring consistency of an inconsistent (partial) instantiation given as an input. The problem arises in three scenarios: (1) the validation task finds an instantiation inconsistent; (2) the completion task detects that a model is consistent, but the given partial instantiation cannot be extended to a complete one; and (3) the model is changed due to new requirements to a configurable product. In order to restore the consistency of an instantiation the reconciliation task comprises two meta-programs.

The first meta-program converts the input OOASP-DDL program into a reified form. This program comprises rules of the form:

$$ooasp\_instantiation(M, I_{new}).$$
$$fact(ooasp(I, \bar{t})) \text{ :- } ooasp(I, \bar{t}).$$

where $ooasp(I, \bar{t})$ stands for one of the OOASP-DDL atoms, listed in Table 2, except $ooasp\_instantiation$. Thus we avoid collisions with previous instantiations I stored in the input DDL program. Instead we generate a fact describing a new instance of a model with a unique identifier I_{new}.

The second meta-program takes the output of the first one as an input and computes a consistent instantiation as well as a set of changes applied to obtain it. The set of changes is obtained by the application of deletion/reuse rules of the

form:

$$1\{reuse(ooasp(I,\bar{t})), delete(ooasp(I,\bar{t}))\}1 \text{ :- } fact(ooasp(I,\bar{t})).$$
$$ooasp(I_{new},\bar{t}) \text{ :- } reuse(ooasp(I,\bar{t})), ooasp\_instantiation(M, I_{new}).$$

A preferred solution can be found if a developer provides costs for reuse/delete actions performed by the reconciliation task.

For example, suppose that the developer created a configuration system that does not implement a domain-specific constraint preventing overheating of the system. Namely, this constraint avoids overheating by disallowing putting two modules of type `ModuleA` next to each other.

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

Due to the added constraint, the instantiation in Figure 3 is no longer valid. The reconciliation task finds a required change by modifying the positions of modules with identifiers `21` and `24`. The result of the reconciliation can be presented to a developer by OOASP framework as shown in Figure 4.

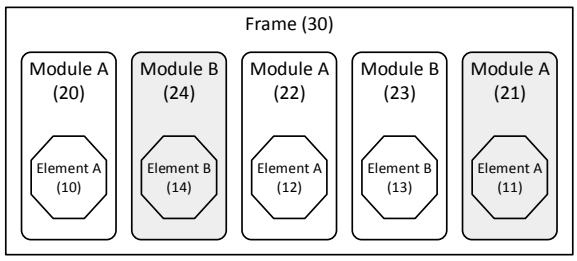

Figure 4: Reconciled configuration for the Modules example

References

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

[2] J. Cabot, R. Clariso, and D. Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80, 2008.

[3] Deepak Dhungana, Andreas A. Falkner, and Alois Haselböck. Generation of conjoint domain models for system-of-systems. In *Generative Programming: Concepts and Experiences*, pages 159–168, 2013.

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

[5] A. Falkner, G. Schenner, G. Friedrich, and A. Ryabokon. Testing Object-Oriented Configurators With ASP. In *ECAI Workshop on Configuration*, pages 21–26, 2012.

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

[7] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.

[8] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in Gringo Series 3. In *Proceedings of the LPNMR*, pages 345–351, 2011.

[9] Martin Gebser, Jörg Pührer, Torsten Schaub, and Hans Tompits. A meta-programming technique for debugging answer-set programs. In *AAAI*, pages 448–453, 2008.

[10] 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.

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

[12] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Catching the Ouroboros: On Debugging Non-ground Answer-Set Programs. *Theory and Practice of Logic Programming*, 10(4-6):2010, 2010.

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

[14] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[15] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL Models Using Boolean Satisfiability. In *Conference on Design, Automation and Test in Europe*, pages 1341–1344, 2010.

[16] Leon S. Sterling and Ehud Y. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT press, 1994.

[17] Markus Stumptner, Gerhard Friedrich, and Alois Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, pages 307–320, 1998.