On Automating Inference of OCL Constraints from Counterexamples and Examples

Duc-Hanh Dang and Jordi Cabot

Abstract. Within model-based approaches, defining domains and domain restrictions for conceptual models or metamodels is significant. Recently, a domain is often presented as a class diagram, and domain restrictions are expressed using the *Object Constraint Language* (OCL). An effective method to define a domain is based on a description of the domain at the instance and example level. So far such a method has often focused on the generation of structure aspects, but have omitted the inference of OCL restrictions that could complement the domain structure and improve the precision of the domain. This paper proposes an approach to automating the inference of OCL restrictions from a domain description in terms of counter- and examples. Candidates are generated by a problem solving, and irrelevant ones are eliminated using the user feedback on generated counter- and examples. Our approach is realized with the support tool InferOCL.

1 Introduction

To capture a domain corresponding to a conceptual model of a system or a metamodel is a significant step to manipulate models within model-driven approaches. An effective approach to define domains is based on a description of the domain at the instance and example level [1]. So far such a method has often focused on the generation of structure aspects captured by a class diagram, but have omitted the inference of restrictions that could complement the domain structure and improve the precision of the domain. The restrictions are often expressed in the Object Constraint Language (OCL) [2]. It puts forward a need to infer OCL restrictions from the instance level information of the domain.

Duc-Hanh Dang
VNU - University of Engineering and Technology, Hanoi, Vietnam
e-mail: hanhdd@vnu.edu.vn

Jordi Cabot
AtlanMod, cole des Mines de Nantes - INRIA, LINA, Nantes, France
e-mail: jordi.cabot@inria.fr

V.-H. Nguyen et al. (eds.), *Knowledge and Systems Engineering,*
Advances in Intelligent Systems and Computing 326, DOI: 10.1007/978-3-319-11680-8_18

The essence of such an inference is to consider the relationship between snapshots (object diagrams) and OCL invariants. Several authors offer methods to check if a snapshot is valid [3]. Current works often translate OCL specifications into other specification environments such as CSP [4], Alloy [5], and relational logic [6] in order to effectively validate snapshots or to find valid (invalid) snapshots or to check properties. The work in [7] proposes a genetic algorithm in order to generate OCL well-formedness rules from counter- and examples. However, the work lacks of considering the relevance of generated candidates.

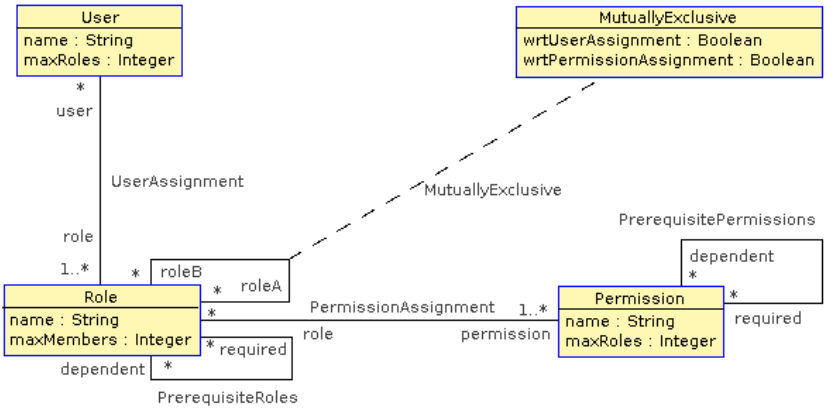
This paper introduces an approach to automating the inference of OCL invariants. At the first step, from input valid and invalid snapshots, OCL invariant candidates are generated using patterns [8]. This paper enhances this step at two points: (1) input snapshots are preprocessed in order to increase the relevance of generated candidates, and (2) the algorithm is improved in order to remove the duplication of candidates. At the second step, *model finding* is employed in order to generate counter- and examples. By getting *user feedback* on generated snapshots, irrelevant cases could be eliminated. We realize the approach with the tool InferOCL, based on the tool EMFtoCSP [4], and the solver ECL^{i}PS^e [9]. We have applied the approach on the domain of Role-Based Access Control (RBAC) models [10]. The experiment shows a possibility to apply the tool InferOCL in practice as well as threads to validity of our approach.

The remainder of this paper is organized as follows. Section 2 motivates the work with a running example. Section 3 introduces an approach to automating the inference of OCL invariants. Section 4 explains our realization for the approach with the tool InferOCL. We present experimental results in Sect. 5 and discusses threats to validity of our approach in Sect. 6. Section 7 surveys related work. This paper is closed with conclusions and an outlook on future work.

2 Running Example

Role-based access control (RBAC) [10] is a popular approach to restricting system access to authorized users. In order to build and integrate RBAC concrete policies in the system, it is necessary to define a conceptual model for it. Figure 1 presents such a conceptual model in form of a class diagram. In this way a RBAC concrete policy would be presented by an object diagram (a snapshot) as depicted in Fig. 2.

Domains like RBAC often require to add certain restrictions on the class model in order to obtain a more precise specification of them. For example, the example RBAC policy shown in Fig. 2 is actually an invalid snapshot since the number of roles that the `ada` could play is greater than the value of the `maxRoles` attribute. Such restrictions are often expressed in OCL and referred to as OCL invariants. We might find the OCL invariants for the simplified RBAC domain [11] as depicted in Fig. 2: (1) The maximum number of roles to which a user is assigned must be less than its attribute `maxRoles`; (2) The assignment of a dependent role with a user postulates the assignment of required roles with the user; (3) A user must not be assigned to both of the mutually exclusive roles.




```
context User inv maxCard_Role: self.role ->size() ≤
self.maxRoles

context User inv requiredInclusion_role: self.role ->forAll(
d | d.required ->forAll(r | self.role ->includes(r)))

context MutuallyExclusive inv retrictedAssoc_user:
self.wrtUserAssignment implies  self.roleA.user
->excludesAll(self.roleB.user)
```

Fig. 1 Simplified RBAC conceptual model

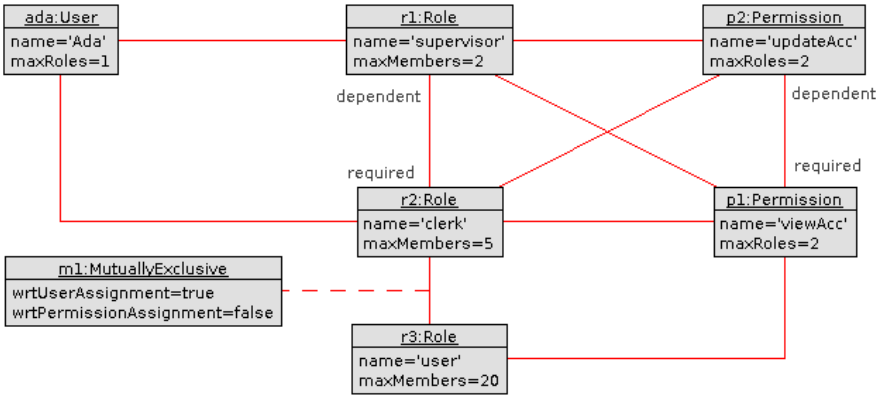


Fig. 2 The instance-level information with snapshots in form of object diagrams

Define OCL invariants by means of examples. To capture such a RBAC domain in a natural way, the modeler often employs snapshots as counter- and examples that provide instance-level information of the domain. Such instance-level information exposes key concepts, relationships, and restrictions of the domain and supports for the modeler grasp them. The challenge is how we can automatically infer OCL invariants from a set of valid and invalid snapshots.

3 A Pattern-Based Automated Inference

Figure 3 illustrates for our approach to automating inference of OCL invariants: *OCL Invariant Patterns* are used as templates to generate OCL invariants that accept valid snapshots and reject invalid ones [8]. The validation of snapshots, that conform to *Conceptual Model*, is determined by the *Domain Knowledge*, i.e., domain experts or model-generating tools such as UML2CSP [4] and USE [3].

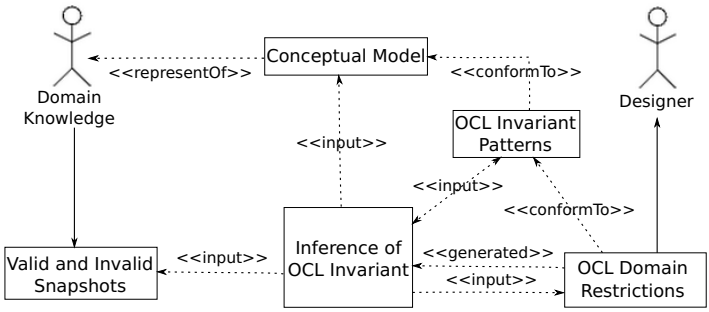


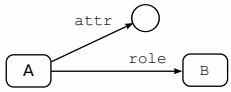
Fig. 3 Overview of the pattern-based approach

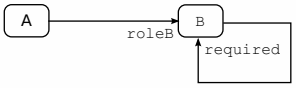
An OCL invariant pattern consists of *an OCL template* and *a matching function* that takes as input a valid snapshot set and an invalid one, binding the variables of the template to values in order to form an OCL invariant [8]. Figure 4 shows the patterns for the OCL invariants within the running example.

3.1 Generating OCL Invariant Candidates as Solving a CSP

We view the generation of OCL invariants as solving a CSP, where OCL invariant patterns are encoded as CSP constraints.

Fetching Patterns from a Catalog. The basic idea is that each OCL invariant set candidate corresponds to a partition of the invalid set $SNOK = \bigcup SNOK_i$, where each invariant is obtained by applying a pattern such that it accepts SOK and rejects $SNOK_i$. We encode the algorithm in prolog as follows:

Pattern	Structure	OCL Template
Maxcard		`context [A] inv:` `self.[role]->size() <= self.[attr]`

RequiredInclusion		`context [A] inv:`
		`self.[roleB]->forAll(d |`
		`d.[required]->forAll( r |`
		`self.[roleB]->includes(r)))`

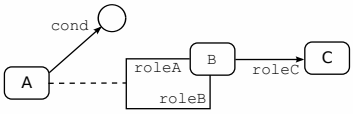
RestrictedAssoc		`context [A] inv:`
		`self.[cond] implies`
		`self.[roleA].[roleC]`
		`->excludesAll(self.[roleB].[roleC])`

Fig. 4 OCL invariant patterns for the running example

```
apply_all(SOK, SNOK, PATTERN, INV) :-
  sort(SNOK, SNOKo),
  %---------- Get all partitions of the invalid snapshot set SNOK -------
  partition(SNOKo, SnokGroups),
  %----------------------------------------------------------------------
  ( foreach(SnokGroup, SnokGroups),
    fromto([], InINV, OutINV, INV),
    param(SOK, PATTERN) do
      member(Pattern, PATTERN),
      %-- Each SnokGroup is rejected by a pattern-generated invariant ---
      applyPattern(Pattern, SOK, SnokGroup, Para),
      %----- Each invariant is captured by a pair (Pattern, Para) -------
      OutINV = [ [ Pattern, Para ] | InINV ]
  ).
```

The aim of the constraint `applyPattern(Pattern,SOK,SnokGroup,Para)` is to define the matching value `Para` as we apply the `Pattern` on the valid snapshots `SOK` and the invalid ones `SnokGroup`. Note that the generated invariant could be formed by the pair `[Pattern,Para]`. We encode the `applyPattern(_,_,_,_)` based on the encoding of each pattern as mentioned below.

Encoding OCL Invariant Patterns. We map each pattern to a CSP constraint so that generated invariants could accept valid snapshots and reject invalid ones. For example, the `maxCard` pattern could be encoded as follows:

```
apply_maxCard(SOK, SNOK, Para):-
  %----------------Ensuring the invariant accepts SOK-------------------
  ( foreach(SnapshotOk, SOK),
    param(Para) do
      maxCard(SnapshotOk, Para, 1)
  ),
  %----------------Ensuring the invariant rejects SNOK------------------
  ( foreach(SnapshotNok, SNOK),
    param(Para) do
      maxCard(SnapshotNok, Para, 0)
  ).
```

The parameterized OCL invariant `maxCard` is translated into the CSP constraint `maxCard(Snapshot,Para,Ret)`, where the `Ret`, that may be 0 or 1, determines the validation of the input snapshot. Such a CSP predicate could be obtained by a translation of OCL invariants into CSP as explained in [4].

3.2 Incorporating User Feedback

In order to capture user's knowledge of the underlying domain in OCL we propose incorporating user feedback at the two aspects: (1) The user could locate part of a snapshot that makes it invalid, and (2) she could determine whether a snapshot is valid or invalid. The first one is to preprocess input snapshots, and the second one is to eliminate irrelevant inferred results.

Preprocessing Input Snapshots. Let us focus on the snapshot shown in Fig. 2, that could be presented as an instance of the following snapshot structure:

```
SnapshotStructure = [ user(oid,name,maxRoles), role(oid,name,maxMembers),
 mutuallyexclusive(oid,roleA,roleB,id,wrtUserAssignment,wrtPermissionAssignment),
 permission(oid,name,maxRoles), userassignment(user,role),
 prerequisiteroles(dependent,required),
 assoccls_mutuallyexclusive(roleA,roleB),
 prerequisitepermissions(dependent,required),
 permissionassignment(role,permission) ]
```

Since this is an invalid snapshot w.r.t the invariant `maxCard_Role` as shown in Fig. 1, the user could highlight relevant part of the snapshot. Technically, we could present preprocessed snapshots in this way:

```
PreprocessedSnapshot = [ [user(1,_,1)],[role(1,_,_),role(2,_,_))],[],[],
 [userassignment(1,1),userassignment(1,2)],[],[],[],[] ]
```

Eliminating Irrelevant Results. We aim to get user feed back in order to eliminate irrelevant generated sets of OCL invariants. The basic idea of this method is that we consider each pair of generated invariant sets, $INV_1 = \{inv_{11}, inv_{12}, ..., inv_{1m}\}$ and $INV_2 = \{inv_{21}, inv_{22}, ..., inv_{2n}\}$, and generating a valid snapshot for the new invariant set $INV_1 \cup \{not\ inv_{2i}\}$, w.r.t each invariant $inv_{2i} \in INV_2$. We then capture user feedback for the generated snapshot example: (1) If the user gives a positive feedback, the result INV_2 as well as the other generated results that reject the snapshot example should be irrelevant; (2) In the other case the result INV_1 and the other generated results that accept the snapshot example should also be irrelevant. In case no valid snapshot w.r.t the new invariant set could be found, we would have $INV_1 \Rightarrow inv_{2i}$, and then if that is true for all $inv_{2i} \in INV_2$, we would have $INV_1 \Rightarrow INV_2$. As we would also have $INV_2 \Rightarrow INV_1$, they are logically equivalent.

4 Tool Support

This section first overviews the support tool InferOCL based on the tool EMFtoCSP [4], and the solver ECL^{i}PS^e [9]. Then, it illustrates the tool and our inference method by applying them for the running example.

4.1 Overview of the InferOCL Tool

Figure 5 outlines the InferOCL tool. First, a model is loaded with an xmi file and encoded in prolog using the EMFtoCSP tool. The control module analyzes input snapshots provided by the user, and sending a query to the ECL^{i}PS^e solver to obtain OCL invariant candidates. In order to generate counter- and examples, the user needs to provide a domain restriction for snapshots.

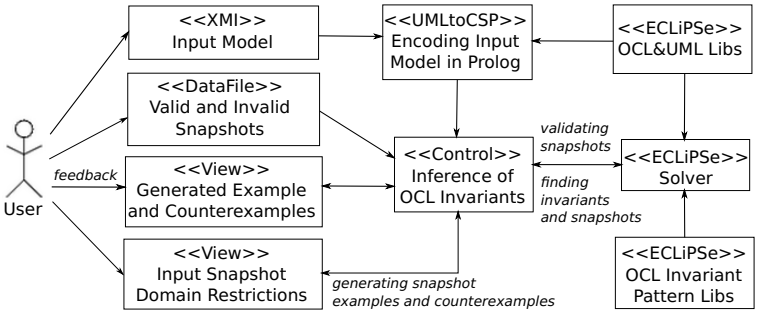


Fig. 5 Overview of the InferOCL tool

The elimination step finishes as no example could be found, and the InferOCL tool brings out final candidates, each of which includes OCL invariants. At this point the candidates are logically equivalent to each other. The user may continue validating the result by examining as much as possible on counter- and examples.

4.2 Applying the Method to the Running Example

We could apply the method to the running example with the following steps.
Step1 - Preprocessing Input Snapshots. The user provides as input 1 valid snapshot and 3 invalid ones that have been preprocessed. The first invalid snapshot corresponds to the `maxCard_Role` invariant, mentioned in Fig. 1. The second and third one corresponds to the `retrictedAssoc_user` and `requiredInclusion_user` invariant, respectively. Note that for sake of clarity, we employ the type Integer instead of the String in this example.

```
SOK = [[[user(1,100,3)], [role(1,100,5),role(2,200,5),role(3,300,5)],
 [mutuallyexclusive(1,1,2,1,0),mutuallyexclusive(2,2,3,1,0)],
 [permission(1,100,2),permission(2,200,2)],
 [userassignment(1,1),userassignment(1,3)], [prerequisiteroles(1,3)],
 [assoccls_mutuallyexclusive(1,2),assoccls_mutuallyexclusive(2,3)],
 [prerequisitepermissions(2,1)],
 [permissionassignment(1,1),permissionassignment(2,1),
  permissionassignment(3,1),permissionassignment(3,2)]]],
SNOK = [[[user(1,_,2)], [role(1,_,_),role(2,_,_),role(3,_,_)],[],[],
 [userassignment(1,1),userassignment(1,2),userassignment(1,3)],[],[],[],[]],
[[user(1,_,_)],[role(1,_,_),role(2,_,_),role(3,_,_)],
```

```
 [mutuallyexclusive(1,1,2,1,_),mutuallyexclusive(2,1,3,1,_),
  mutuallyexclusive(3,2,3,1,_)],[],[userassignment(1,1),userassignment(1,3)],[],
 [assoccls_mutuallyexclusive(1,2),assoccls_mutuallyexclusive(1,3),
  assoccls_mutuallyexclusive(2,3)],[],[]],
[[user(1,_,_)],[role(1,_,_),role(2,_,_)],[],[],[userassignment(1,1)],
 [prerequisiteroles(1,2)],[],[],[]]]
```

Step 2 - Generating Candidates. With the input data from Step 1, the InferOCL tool could generate 18 candidates by an inference on our experimental catalog of OCL invariant patterns, including the 3 patterns as depicted in Fig. 4 and the 5 patterns as introduced in [8]. For example, the following is one of the candidates, that includes three OCL invariants.

```
context Role inv  cardInv_roleB: self.roleB -> size()
< 2

context User inv  requiredInclusion_role: self.role
-> forAll(d | d.required -> forAll(r | self.role
-> includes(r)))

context User inv  intv_maxRoles: 2 < self.maxRoles
```

Step 3 - Eliminating Irrelevant Candidates. The elimination of irrelevant candidates for the running example is summarized as in Table 1. At the first step the user provides a domain restriction for snapshots, that includes bounds for the size of classes and associations and attribute values:

```
User :: 0..10; name: Integer :: 1..10; maxRoles: Integer :: 1..10;
Role :: 0..10; name: Integer :: 1..10; maxMembers: Integer :: 1..10;
MutuallyExclusive :: 0..10; wrtUserAssignment: Boolean :: 0..1;
                            wrtPermissionAssignment: Boolean :: 0..1;
Permission :: 0..10; name: Integer :: 1..10; maxRoles: Integer :: 1..10;
UserAssignment :: 0..10; PrerequisiteRoles :: 0..10;
PrerequisitePermission :: 0..10; PermissionAssignment :: 0..10;
```

The generated example at step 2 in Table 1 is irrelevant w.r.t the `retrictedAssoc_user` restriction. The irrelevant snapshots at step 3 and 5 correspond to the `requiredInclusion_user` and `maxCard_Role` restriction.

Step 4 - Examining on Equivalent Candidates. At this step the InferOCL tool brings out the final candidate including 3 OCL invariants as mentioned in Fig. 1. The user could continue to validate this result and each equivalent candidate in general by examining on generated counter- and examples.

5 Experimental Results

This experiment is performed on a 64-bit computer with 2.70GHz Intel Core i7 processor and 8Gb RAM. We take as input 05 arbitrary snapshots for the valid set *SOK* and from 3 to 11 other ones for the invalid set *SNOK*. In the first case the input *SNOK* is preprocessed and the other case without preprocessing. The result of generating candidates and the corresponding performance time is presented as in Table 2. We could realize that the number of generated candidates decreases as

Table 1 Eliminating irrelevant candidates for the running example

Step	Generated Example Snapshot	Feedback	Candidate
1	[[],[role(1,1,1),role(2,1,1)],[mutuallyexclusive(1,1,1,0,0), mutuallyexclusive(2,2,1,0,0)],[permission(1,1,1)],[],[], [assoccls_mutuallyexclusive(1,1), assoccls_mutuallyexclusive(2,1)],[], [permissionassignment(1,1),permissionassignment(2,1)]]	Yes	12
2	[[user(1,10,3)],[role(1,1,1),role(2,1,1)], [mutuallyexclusive(1,1,1,1,0)],[permission(1,1,1)], [userassignment(1,2),userassignment(1,1)],[], [assoccls_mutuallyexclusive(1,1)],[], [permissionassignment(1,1),permissionassignment(2,1)]]	No	6
3	[[user(1,10,3)],[role(1,1,1),role(2,1,1),role(3,1,1)],[], [permission(1,1,1)],[userassignment(1,2), userassignment(1,1)],[prerequisiteroles(1,3)],[],[], [permissionassignment(1,1),permissionassignment(2,1), permissionassignment(3,1)]]	No	3
4	[[user(1,10,3)],[role(1,1,1),role(2,1,1),role(3,1,1)],[], [permission(1,1,1)],[userassignment(1,3), userassignment(1,2),userassignment(1,1)],[],[],[], [permissionassignment(1,1),permissionassignment(2,1), permissionassignment(3,1)]]	Yes	2
5	[[user(1,10,3)],[role(1,1,1),role(2,1,1),role(3,1,1), role(4,1,1)],[],[permission(1,1,1)],[userassignment(1,4), userassignment(1,3),userassignment(1,2), userassignment(1,1)],[],[],[],[permissionassignment(1,1), permissionassignment(2,1),permissionassignment(3,1), permissionassignment(4,1)]]	No	1

the input is preprocessed, i.e., the preprocessing highlights invalid part of snapshots so that irrelevant cases are eliminated. The generation of candidates is also faster performed as the input is preprocessed.

The experiment points out a certain restriction on the size of input data: The tool works well on 10 invalid snapshots of a conceptual model including 4 classes and 5 associations. It could accept a large domain restriction in order to generate counter- and examples. Specifically, with a quite large domain and applicable in practice like 0..500 for attribute values and a simple domain for the number of classes and associations like 1..10, the finding is performed just in few seconds.

Table 2 The number of generated candidates and the performance time in two cases, (1) with preprocessing and (2) without preprocessing

SNOK Size	Candidates 1	Candidates 2	Time 1	Time 2
3	1	2	0.31s	0.29s
4	1	6	0.78s	0.92s
5	1	6	2.09s	2.55s
6	2	18	8.05s	10.55s
7	4	54	34.62s	46.32s
8	4	54	152.72s	210.77s
9	4	108	738.04s	996.49s
10	8	324	4077.09s	5244.91s
11	8	324	23030.21s	30252.04s

6 Threats to Validity

There are certain threats to validity of our inference approach. First, the approach is just evaluated on simple examples. Further explorations on larger case studies need to be performed. Second, the method to generate invariant candidates currently retains limitations in performance. We would need a better preprocessing solution, e.g., a web-based solution, to effectively get user feedback on a large number of input snapshots. Last but not least, the current method to eliminate irrelevant candidates puts forward challenges: (1) How we can always confirm whether a counter- or an example exists on a given domain, and (2) how a user-given domain restriction could be defined with less effort from the user.

7 Related Work

OCL Inference. The need to infer OCL restrictions from instance-level information has been considered in the work [12], where a technique to infer metamodels from instances is introduced. The work in [7] proposes an approach based on genetic programming. They encode sets of invariants as populations, and genetic operators is provided in order to evolve the population and to obtain the final generation that accepts examples and rejects counterexamples. However, the work does not consider the relevance of generated candidates.

OCL and Snapshot. There is significant work concentrating on the relationship between OCL constraints and snapshots. In [3] the USE tool allows us to check OCL restrictions on class diagrams. The paper in [13] proposes to validate UML and OCL models by generating snapshots from a declarative description. Other works often

translate the OCL specification into specification environments such as CSP [4], Alloy [5], and relational logic [6] in order to effectively validate snapshots or to find valid (invalid) snapshots or to check properties. We here focus on how OCL constraints are defined by analyzing snapshots.

OCL Pattern. Our method to specify OCL invariant pattern is related to the paper in [14], in which a method to represent OCL expressions in a SBVR form in order to generate natural language explanations for business rules is proposed. The paper in [15] introduces a method to generate semantically equivalent alternatives for the initially defined OCL constraints. It assists the designer to better define OCL constraints.

OCL Learning. The essence of this work is a concept learning, an issue in machine learning. The paper in [16] introduces the L* algorithm in order to generate the best assumption in form of deterministic final-state automata from examples. The paper in [17] proposes a SAT-based method to acquire binary constraint networks. The paper in [18] discusses a method to generate OCL constraints from natural language. Our work focuses on learning OCL constraints.

This work continues our previous work in [8]. In this work a new algorithm to fetch patterns is introduced so that we could get over the duplication of inferred results. Moreover, this work supports for multiple restrictions instead of only one as explained in the previous work. The proposal in this work to incorporate user feedback could help decreasing the number of generated candidates. This work also offers a more effective strategy to eliminate irrelevant cases.

8 Conclusions

This paper has introduced an approach to automating the inference of domain restrictions as OCL invariants from the instance-level information captured by valid and invalid snapshots. In the language engineering context the underlying domain classifies models, while within the software engineering context the domain corresponds to a conceptual model, representing for system snapshots. The basic idea of this approach is to employ OCL invariants patterns as inference patterns in order to infer invariant candidates from a set of valid and invalid snapshots. The user could help to preprocess input snapshots so that less irrelevant candidates are generated. Getting user feedback on generated snapshots also help to remove irrelevant candidates. The approach is realized with the InferOCL tool, based on the model finding tool EMFtoCSP and the ECL^{i}PS^e solver. The experiment on the simplified RBAC model shows a possibility of the InferOCL tool in practice as well as threads to validity of the approach.

In future, we aim to apply the approach on larger case studies in order to get detailed feedback on it. Such a task could be to validate and to infer invariants for the UML metamodel. It would require us to enrich and maintain the catalog of patterns as well as to extend the method to generate invariant candidates and to eliminate irrelevant ones. To enhance the InferOCL tool with such new features is also on

the focus of our future work. Our enhancements in model finding would be further assessed in order to make contribution to that issue.

Acknowledgement. This work was supported by the research project QG.14.06, Vietnam National University, Hanoi. We also thank anonymous reviewers for their comments on the earlier version of this paper.

References

[1] Sutcliffe, A.G., Maiden, N.A.M., Minocha, S., Manuel, D.: Supporting Scenario-Based Requirements Engineering. IEEE Trans. Software Eng. 24(12), 1072–1088 (1998)
[2] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley Professional (2003)
[3] Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69(1-3), 27–34 (2007)
[4] Cabot, J., Claris, R., Riera, D.: UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming. In: Kurt Stirewalt, R.E., Alexander Egyed, B.F. (eds.) Proc. 22th Int. Conf. Automated Software Engineering (ASE), pp. 547–548. ACM, New York (2007)
[5] Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)
[6] Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 415–431. Springer, Heidelberg (2012)
[7] Faunes, M., Cadavid, J.J., Baudry, B., Sahraoui, H.A., Combemale, B.: Automatically searching for metamodel well-formedness rules in examples and counter-examples. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 187–202. Springer, Heidelberg (2013)
[8] Dang, D.H., Cabot, J.: Automating Inference of OCL Business Rules from User Scenarios. In: Proc. 20th Asia-Pacific Conf. Software Engineering (APSEC), pp. 156–163. IEEE (2013)
[9] ECLiPSe: The ECLiPSe Constraint Programming System. Version 6.1 (June 2013)
[10] Ferraiolo, D., Kuhn, D.: Role-Based Access Control. In: Proc. 15th National Computer Security Conf., pp. 554–563 (1992)
[11] Kuhlmann, M., Sohr, K., Gogolla, M.: Comprehensive Two-Level Analysis of Static and Dynamic RBAC Constraints with UML and OCL. In: Baik, J., Massacci, F., Zulkernine, M. (eds.) Proc. 5th Int. Conf. Secure Software Integration and Reliability Improvement (SSIRI), pp. 108–117. IEEE (2011)
[12] Javed, F., Mernik, M., Gray, J., Bryant, B.R.: MARS: A Metamodel Recovery System Using Grammar Inference. Information & Software Technology 50(9-10), 948–968 (2008)
[13] Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Software and System Modeling 4(4), 386–398 (2005)
[14] Pau, R., Cabot, J.: Paraphrasing OCL Expressions with SBVR. In: Kapetanios, E., Sugumaran, V., Spiliopoulou, M. (eds.) NLDB 2008. LNCS, vol. 5039, pp. 311–316. Springer, Heidelberg (2008)

[15] Cabot, J., Teniente, E.: Transformation Techniques for OCL Constraints. Science of Computer Programming 68(3), 152–168 (2007)
[16] Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75(2), 87–106 (1987)
[17] Bessière, C., Coletta, R., Koriche, F., O'Sullivan, B.: A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 23–34. Springer, Heidelberg (2005)
[18] Bajwa, I., Bordbar, B., Lee, M.: OCL Constraints Generation from Natural Language Specification. In: Proc. 14th Int. Conf. Enterprise Distributed Object Computing Conference (EDOC), pp. 204–213. IEEE (2010)