A Pattern-matching based Approach for Problem Solving in Model Transformations

Youness Laghouaouta, Pierre Laforcade and Esteban Loiseau
Le Mans Université, LIUM, EA 4023, France
{youness.laghouaouta, pierre.laforcade, esteban.loiseau}@univ-lemans.fr

Keywords: Model Driven Engineering, Model Transformation, Pattern Matching, Constraint Satisfaction Problem.

Abstract: As MDE (Model Driven Engineering) principles are increasingly applied in the development of complex software, the use of constraint solving in the specification of model transformations is often required. However, transformation techniques do not provide fully integrated supports for solving constraints, and external solvers are not well adapted. To deal with this issue, this paper proposes a pattern-matching based approach as a promising solution for enforcing constraints on target models. A transformation infrastructure is semi-automatically generated and it provides support for specifying patterns, searching for match models, and producing valid target models. Finally, a use case is presented in order to illustrate our contribution.

1 INTRODUCTION

Model Driven Engineering (MDE) proposes models to represent all artifacts handled by a software development process. The principle is to raise the abstraction level by using models as first class entities in dedicated model management operations (e.g. model transformation, model composition, model validation...).

Among these operations, the transformation of models is the most important and well-studied operation. It allows the automatic generation of target models from source ones. These models conform to metamodels that define the structure and well-formedness rules. Besides, a transformation specification/definition includes descriptions of how constructs of source metamodels can be transformed into constructs of target metamodels (Mens and Gorp, 2006).

Several related techniques have been proposed in the literature (Jouault and Kurtev, 2005)(Kolovos et al., 2008)(OMG, 2008). Despite their differences (e.g. textual or graphical concrete syntax; declarative, imperative or hybrid nature; rule-based or relation-based languages...), their shared goal is to provide developers with supports to specify the production of valid target models from source ones (i.e. generally higher-level models).

A target model is valid when it conforms to the structuring defined by its metamodel and satisfies all the related constraints. In practice, constraints cannot be expressed by means of metamodel constructs and implies the use of additional formalisms (e.g. OCL (OMG, 2014)). Likewise, model transformation techniques are not well supported for enforcing all constraints, and usually developers require using external constraint solvers or libraries. However, expressing a Constraint Satisfaction Problem (CSP) is a complex and error prone activity because developers are faced with the domain divergence of the managed elements (i.e. model elements in model transformations versus numeric values in constraint solvers).

Our contribution is then a practical constraint solving approach for model transformations. The objective is to allow enforcing constraint on target models while simplifying the expression of the constraint based problem. To this aim, a CSP is considered as a pattern matching problem specified by means of model elements. Besides, the pattern matching is included in a global process that allows producing the expected target models of a given transformation scenario.

The remainder of this paper is structured as follows. In Section 2, we present the context of our proposal and motivate the need for a practical constraint solving approach for model transformations. Section 3 gives an overview of the main concepts of our approach as well as relevant implementation details. In section 4, we demonstrate the soundness of our approach using an illustrative transformation scenario. Section 5 lists related work. Finally, Section 6 summarizes this paper and presents future work.

2 MOTIVATION

This work has been motivated by the need for a framework that eases the design and the validation of a serious game for helping children with ASD (Autistic Syndrome Disorder) to improve their visual skills. Given the specific needs of the players, it is predominant to take into account the domain experts' directions and requirements during the design phase. This includes the domain elements, and the rules involved during the generation of adapted learning scenarios.

MDE provides principles for conducting collaborative design and validation sessions between computers scientists and domain experts. It allows representing the domain elements as active models (i.e. children profiles, game components and game scenarios). As a consequence, the scenarization process can be dynamically generated by means of model transformations (i.e. the other alternative would be to design and implement all possible configurations of scenarios). Indeed, the profile model and game component model can be transformed to produce adapted scenarios. The latter are used to validate the generation rules and will subsequently form a basis for real exploitation within the game.

In previous work (Laforcade et al., 2018), we have proposed a metamodel for structuring all the dimensions related to the game. As for the generation of scenarios adapted to children profiles, it is implemented as a model transformation written in Java/EMF (Steinberg et al., 2009). Certainly, the proposed implementation allows optimizing the validation task in the sense that game scenarios are generated in demand and without additional effort. However, the problem arises when domain experts propose modifications to the generation rules.

Indeed, it is not easy to identify the transformation fragments impacted by the expressed changes. The way the transformation is specified does not reveal matches between each experts direction/requirement (i.e. considered here as a constraint) and the transformation fragments that allow building conformed models. In addition, the experts directions/requirements are not easy to implement even when the transformation is specified from scratch. Several constraints are expressed in order of priority. They are global constraints as they are attached to a set of model elements and not to separated ones. In fact, the proposed model transformation uses an external constraints solving library to tackle some very specific generation steps.

We conducted past experiences with domain experts using the aforementioned implementation. It allowed us to collect feedbacks and drew two conclusions. MDE principles ease the co-design and validation tasks (i.e. by structuring the domain elements and allowing the automatic generation of adaptive scenarios). However, the way to implement the production of learning scenarios is problematic (i.e. especially when changes are expressed). A first way to deal with this issue consisted in testing other languages/supports for model transformations. Concretely, we have exploited the ETL language (Kolovos et al., 2008) and the meta-language *Melange* (Degueule et al., 2015) for expressing generation of scenarios.

Although ETL allows specifying the transformation in a much more structured way compared to Java/EMF (e.g. rules, operations, pre and post blocks), the lack of a CSP (Constraint Satisfaction Problem) support raises a significant issue. As for *Melange* (i.e. a language workbench that allows expressing operational semantics by augmenting meta-classes with behaviors), the generation concern is specified in a modular manner which helps to identify the components (e.g. meta-classes, operations) related to the expressed change. Also, it is possible to reuse existing Java libraries for CSP solving. However, like the Java/EMF transformation, it is not easy to express the direction/requirement of the expert by means of CSP. This is due to the divergence between domains of values supported by the CSP solver (essentially integer and real values) and the concerned model elements.

Accordingly, we have set three requirements for the design and validation framework discussed above. Even if we are focusing on the development of the presented serious game, we think that the application scope of such framework can be extended to software/systems with complex and variant constraints. Our goal is to provide support for:

1. the generation of target models by transforming source ones;
2. the specification of constraints applied to target models in a simple manner and constraint solving;
3. the modification or reconfiguration of the transformation in case of constraint changes.

The next section details our proposal for a model transformation approach that facilitates the expression of problems addressed by constraints satisfaction (objectives 1 and 2). This proposal fits into our co-design and validation framework. Indeed, it was designed in order to meet the third objective as well. However, details concerning the modification/reconfiguration of the transformation when changes occur are out of the paper scope.

3 PATTERN MATCHING FOR CSP SOLVING IN MODEL TRANSFORMATIONS

This section explains how a model transformation implying constraint solving could be considered as a pattern matching problem. First, we present a global overview and give the principles of our approach. Thereafter, we present the configuration metamodel that defines the relevant concepts. Finally, we detail the generation process of the infrastructure supporting the proposed approach.

3.1 Global Overview

Basically, a CSP is defined as a set of variables, variable domains (i.e. possible values for each variable) and a set of constraints. A solution is an assignment of values to each variable that satisfies every constraint. As for graph pattern matching, it is based on *(sub)graph isomorphism* and requires finding an image (i.e. match) of a given graph (i.e. pattern graph) in another graph (i.e. source graph) (Larrosa and Valiente, 2002).

In the literature, several work address the joint use of CSP and pattern matching (Rudolf, 1998)(Larrosa and Valiente, 2002)(Taentzer et al., 1999). Essentially, the graph pattern matching is expressed and resolved as a CSP. The aim is to improve matching performance by exploiting the rich and advanced research work done in the CSP field. In order to obtain the CSP equivalent of a pattern matching problem, some matches have been established between concepts of the two domains (Rudolf, 1998):

- CSP variables correspond to the objects of the pattern graph;
- variable domains correspond to the source graph objects to be matched into;
- constraints correspond to the restrictions that apply to a graph morphism.

Our approach is based on a reverse use of these matches. The core principle is to consider a pattern matching problem as a high level specification of a CSP. Hence, a CSP problem over a model can be directly expressed by means of model elements rather than establishing non evident matches to basic variable domains supported by CSP solvers (e.g. integers, reals).

Figure 1 gives a global overview of our approach. A model transformation that implies constraint solving is decomposed into two steps: a pattern matching step and a transformation step. The idea is to express all constraints to enforce on target models through a relevant pattern. The found match is then transformed into valid target models. Therefore, the expression of the pattern has to consider the following requirements:

- although the pattern is expressed by means of source model elements, it has to ensure the satisfaction of all constraints related to target models;
- a match model has to be sufficient enough to ensure a complete generation of target models.

Furthermore, the expression of a pattern is decomposed into two parts. The structure part refers to the elements to be matched into source models, while the constraint part refers to the different constraints that force the identification of a match model. This decoupling makes it possible to associate multiple constraints (i.e. classed by order of priority) to the same pattern. Also, variation of a constraint does not affect the transformation because this latter is specified using the pattern structure.

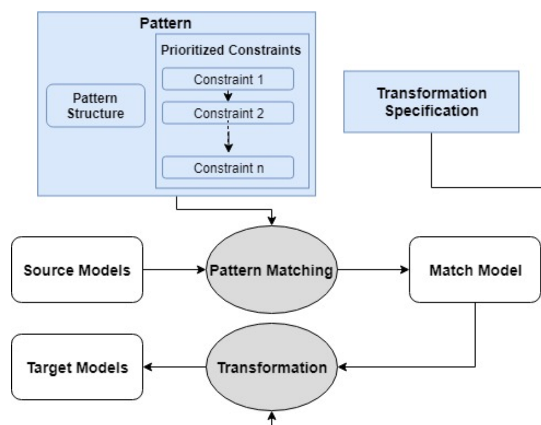

Figure 1: Global overview of the transformation process.

Figure 2 illustrates our proposal. The source model contains a set of squares with a colored background. Each one contains numbered triangles of different areas. The transformation scenario consists of piling up 4 colored triangles. Each target triangle results from the transformation of a source one while preserving the same area. As for its background color, it corresponds to the color of the square container of the source counterpart. Two constraints have to be satisfied by the target model:

- the area of the contained triangle must be less than the container one in order to produce a coherent piling up;
- two adjacent triangles must have different colors.

For this transformation scenario, the pattern structure consists of an ordered set of four triangles. Each

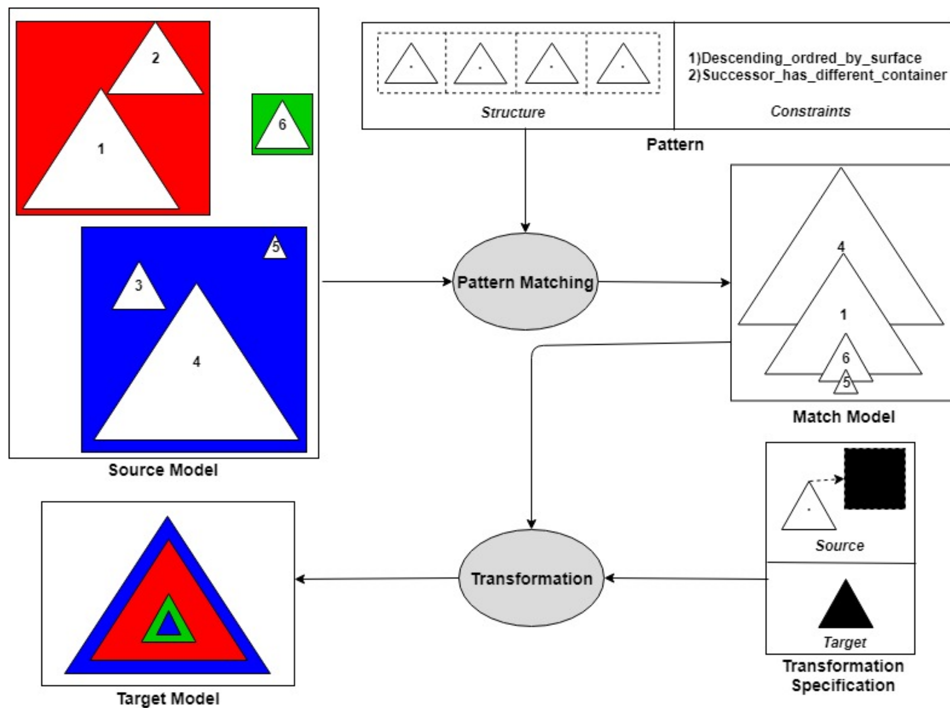

Figure 2: Simple application example.

one can match one of the source triangles. As for the constraint part, we specify that a match is valid if the matched triangles are in descending order of area. Furthermore, successive triangles must belong to different squares. Once a valid match occurs, the transformation is applied on each matched triangle for copying it and assigning the background color of its container. We have to note that squares are not matched by the pattern. They are derived from matched triangles.

3.2 Configuration Metamodel

As discussed before, the pattern specification (i.e. structure and constraints parts) underpins the proposed transformation approach. The relevant information is considered as a configuration for generating the transformation infrastructure. It is stored in a model which conforms to the metamodel depicted in Figure 3.

A configuration references different models (i.e. source models, target models and the implied metamodels). As for the pattern structure, a configuration is defined by multiple role types. Each one can be considered as an abstraction of a set of concrete roles (i.e. pattern elements used to match source model elements) sharing same characteristics or managed as a set. Indeed, a *RoleType* is characterized by the number of concrete roles *nbRoles* and references

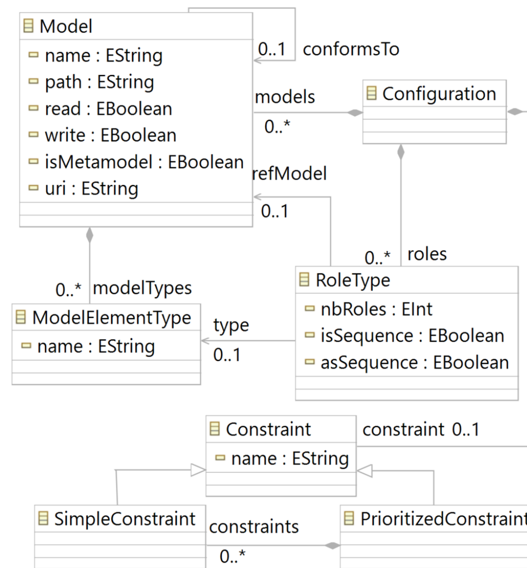

Figure 3: Configuration metamodel.

a specific source model element type. For example, the pattern depicted in Figure 2 can be expressed by one *RoleType* instance. This latter references the model element type corresponding to triangles and has an *nbRoles* equals to four. Therefore, the declared role type is an abstraction of four concrete roles and each of them is used to match a unique triangle of the source model.

Besides, the *isSequence* attribute is used to express that a role type must match a sub-set of source elements (i.e the same element cannot be matched multiple times). While the *asSequence* attribute allows grouping multiple elements matched by the corresponding roles in order to be managed as one element (i.e. essentially for optimizing the validation and transformation tasks).

As for to the pattern constraint part, a configuration expresses if a match model is validated against one constraint level (*SimpleConstraint*) or multi-levels and prioritized constraints (*Prioritized-Constraint*). A constraint is characterized by a name that gives an idea of the validation logic. One can note that the complete constraints specification (i.e. by means of conditions for example) is not covered by the proposed metamodel. Indeed, the configuration model does not ensure the generation of the entire transformation infrastructure. This latter includes resources that have to be manually completed by developers. The next section details these aspects by presenting the infrastructure generation process.

3.3 Infrastructure Generation Process

The first step to produce transformation infrastructure is the generation of the configuration (see Figure 4). For that, we provide developers with a GUI allowing them to specify all paths of the managed models and metamodels to which they conform. A configuration model can then be automatically generated. It includes the input information as well as other automatically derived data (e.g. metamodels URIs, model elements types).

As previously stated, the configuration model have to be completed with the pattern structure as well as with the constraints type (i.e. simple or hierarchical). To make this task easier for developers, we have associated a concrete textual syntax to the configuration metamodel and implemented a dedicated XText editor (Bettini, 2016).

Once the configuration is completed, developers can then ask for the automatic generation of the transformation infrastructure. This is concretely done by associating a specific EPL pattern (Kolovos and Paige, 2017) to each constraint level (i.e. in case of hierarchical constraints). EPL is a language that provides support for the specification and detection of structural patterns in models that conform to diverse metamodels (Kolovos and Paige, 2017). Essentially, an EPL pattern consists of a set of typed roles used to capture adequate combinations from source models and a match condition to evaluate the validity of a combination. In our case, typed roles are derived from *RoleType* instances (i.e. with respect to *nbRoles*, *type* and *refModel* values) while the match condition is viewed as a Boolean operation that references a considered constraint.

Besides, the sequencing of the patterns execution is described as an ANT-based Epsilon workflow (Kolovos et al., 2017). For each EPL pattern, a dedicated target and task pair is generated. Besides, *depends* properties of each generated target are specified in order to prohibits the execution of a successor pattern (i.e. with respect to constraint priorities) if a match has already been found for the current pattern.

These details are hidden to developers by separating the patterns (i.e. generated automatically) from some required resources to be completed. Since all the patterns have the same structure (i.e. derived from the configuration model), a developer needs to provide only one specification for transforming match models. The transformation is specified by means of EOL operations (Kolovos et al., 2006). Regarding constraints, they are expressed in the validation resource. Indeed, for each constraint, an EOL operation is generated according to pattern roles. Each of them must be implemented by the developer in order to express the validation logic (i.e. when a model captured by pattern roles is considered to be a valid match).

Domain restriction resources make it possible to refine the constraints specification. Unlike the validation which applies on an entire match model, the restriction concerns only one single role (e.g. do not capture a triangle if its area exceeds a threshold). Finally, the remaining resource allows the developer to assign operations to meta-classes. These operations can be called by other resources.

The way in which the transformation infrastructure is structured brings further benefits. When a constraint changes, the transformation resource is not impacted. Besides, the operations associated to the implied meta-classes can be reused when changing the pattern structure or the transformation scenario as long as the same metamodels are involved.

4 APPLICATION

This section is dedicated to the application of the proposed approach. First, we briefly present the serious game that motivates the overall proposal and we describe the selected use case. Then, we illustrate each step of the process of generating the transformation infrastructure.

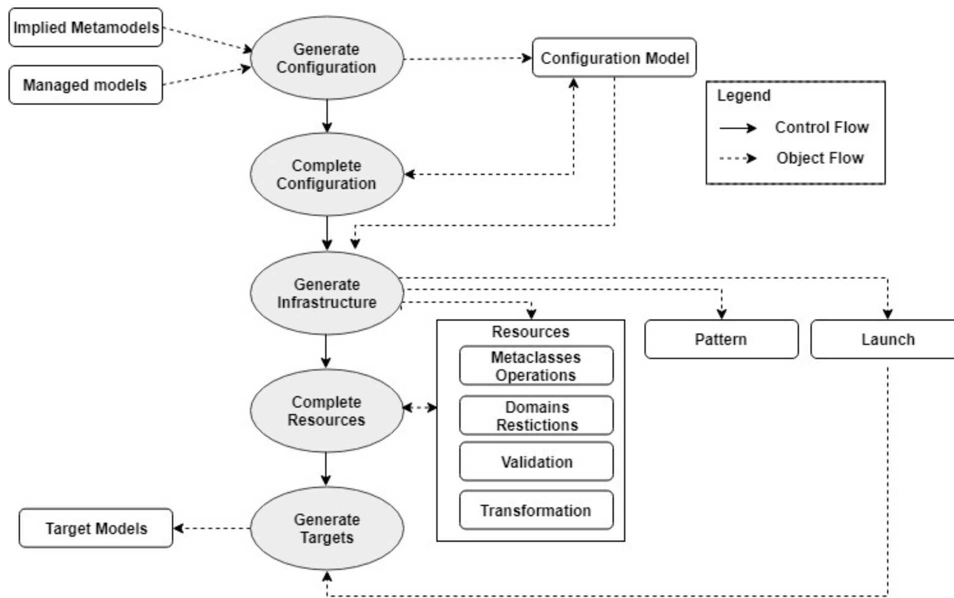

Figure 4: Process for generating the transformation infrastructure.

4.1 Use Case

The application context is the development of a mobile learning game dedicated to children with ASD (Autistic Syndrome Disorder). The game intends to support the learning of visual skills. This game is based on mechanics from "escape-room" games: the player's goal is to open a locked door to escape the room; to this end, the user has to solve numerous puzzles often requiring observation and deduction. The mobility feature will allow the learning to take place wherever the parents, therapeutics, or child wants it.

The global domain elements required for the generation of game sessions are structured into three related parts: game description elements, profile-related elements, and scenario elements. The required constructs have been defined by a dedicated metamodel.

- game description model: it describes all the game elements (e.g. skills, resources or exercisers, in-game objects...);
- profile model: it describes a user profile;
- scenario model: it describes the targeted learning objectives, the selected exercises and involved resources; it is generated from the two first models.

In addition, this scenario is decomposed into three different scenarios:

- objective scenario: it references the selected visual performance skills in accordance to the current child profile.
- structural scenario: it refers to the various scenes where game levels will take place. This scenario extends the previous one. It is generated from knowledge domain rules stating the relations between scenes and the targeted skills they can deal with.
- features scenario: it expresses the additional inner-resources/fine-grained elements to be associated to each selected scene (e.g. objects appearing in a scene, their positions...). The features scenario includes components of previous scenarios. It specifies the overall information required by a game engine to drive the set-up of a learning game session.

For convenience, the proposed use case is limited to the generation of the higher-level scenario (i.e. objective scenario). The two input models (i.e. game description and profile) conform to the metamodel depicted in Figure 5. It is worth noting that the presented metamodel is just an excerpt of the global one that defines all the game constructs.

The game description model has been specified on the base of experts' requirements. It expresses four visual performance skills B3-B4-B8-B25 (respectively matching object to object, matching object to image, sorting categories of objects, making a seriation) and their dependency relations. Figure 6 shows that the B3 skill unlocks the B4 and B8 skills (i.e. completing B3 at its highest difficulty allows children to progress independently with the learning of the B4 and B8 skills).

A fictive child profile model (see Figure 7) expresses that the B3 skill is acquired at its highest level. The B4 skill is at the elementary level, B8 is at the

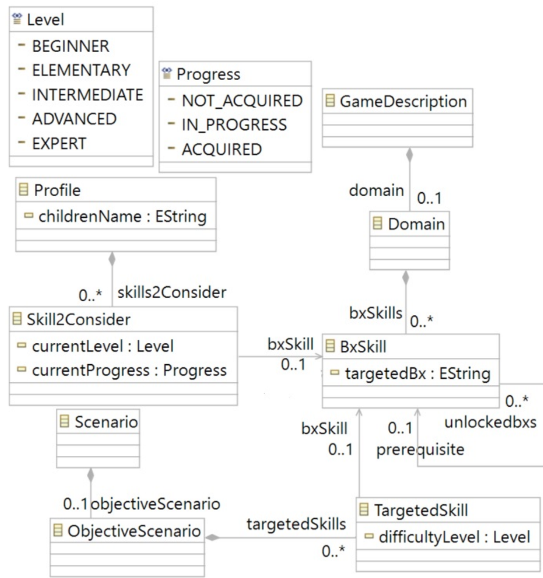

Figure 5: Objective scenario metamodel.

expert level but is not acquired by the child, and B25 is at the beginner level. Note that we have used the Emf2gv[1] project to provide a user friendly representation of the managed models.

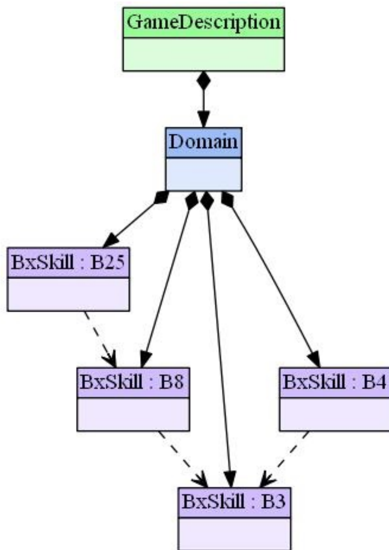

Figure 6: Game description model.

Regarding the generation of the objective scenario, the expert has expressed four constraints to enforce on the output model:

- **Constraint 1:** it has to include exactly four targeted skills.
- **Constraint 2:** targeted skills are proposed among the profile skills for which the child has acquired

[1]See http://sourceforge.net/projects/emf2gv.

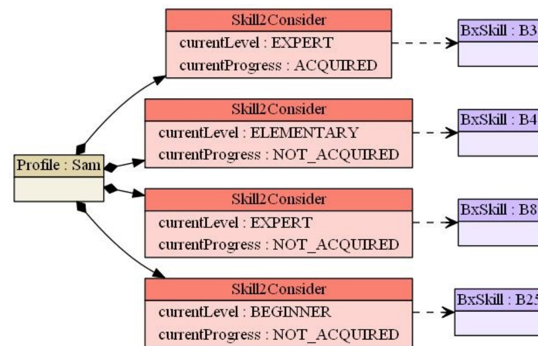

Figure 7: Profile model.

the corresponding prerequisites and did not reach a maximum level.

- **Constraint 3:** it would be best to propose different targeted skills.
- **Constraint 4:** if constraint 3 fails, the objective scenario must be arranged in view of excluding successor elements referencing the same skill.

4.2 Infrastructure Generation

In order to perform the described transformation scenario, we start by generating the relevant configuration. This latter can be completed by defining the pattern structure and identifying constraints through a dedicated Xtext editor.

For the presented use case, the pattern comprises five roles in order to satisfy constraint 1: one *Profile* and four *Skill2Consider* elements. Indeed, the matched *Skill2Consider* elements are viewed as the source equivalents of the targeted skills to be generated. As for constraints 3 and 4, they are expressed as prioritized constraints (see Figure 8) while greatest priority is given to the first declared constraint. Aside from the pattern structure and constraints parts, all the other elements are automatically generated.

Once the configuration model is completed, the transformation infrastructure can be generated (see Figure 9). Recalling from Section 3.3, an EPL pattern is automatically generated for each constraint level and the related details are hidden to developers by separating patterns and the required resources. For the application example, two EPL patterns are generated respectively for constraints 3 and 4. Listing 1 illustrates an excerpt of the pattern generated for constraint 3 (i.e. proposing different targeted skills).

Regarding the operations resource, for each model element type an EOL script is provided to allow defining specific operations. As for the domain restriction resource, it is possible to specify guard conditions in order to restrict the possible elements to be

```
Configuration
{
Models:
    metamodel "mmcs" "C:/Eclipse/workspace/ICSOFT/models/MMCS.ecore"
        uri: "http://mmcs/1.0"
    Types:
        "GameDescription"
        "Domain"
        "BxSkill"
        "Skill2Consider"
        "Profile"
        "TargetedSkill"
        "ObjectiveScenario"
        "Scenario"
    model "m1" "C:/Eclipse/workspace/ICSOFT/models/Profile.xmi" read mmcs
    model "m2" "C:/Eclipse/workspace/ICSOFT/models/GameDescription.xmi" read mmcs
    model "m3" "C:/Eclipse/workspace/ICSOFT/models/Scenario.xmi" write mmcs
Pattern:
    roleType m1!mmcs.Profile [1]
    roleType m1!mmcs.Skill2Consider [4]
Constraint:
    SimpleConstraint "allDifferent"
    SimpleConstraint "followingNotMatch"
}
```

Figure 8: Configuration model.

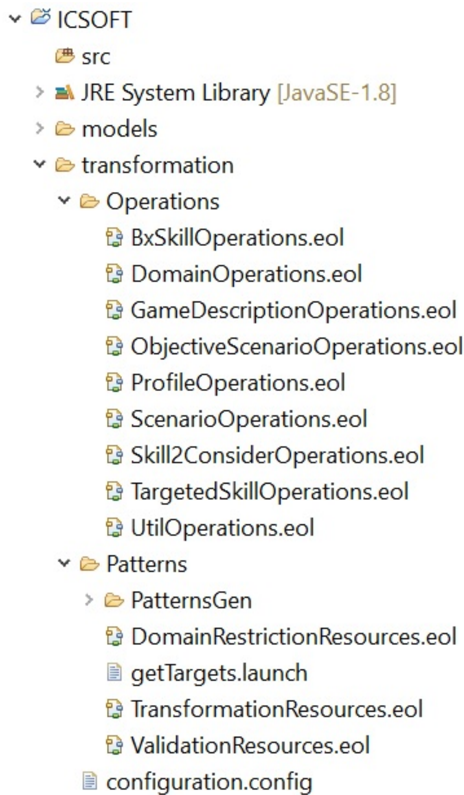

Figure 9: Transformation infrastructure.

```
pattern Pattern
        r0 : m1!Profile
                guard : r0.ProfileDomainRestriction(),
        r1 : m1!Skill2Consider
                guard :
                        r1.Skill2ConsiderDomainRestriction(),

        r2 : m1!Skill2Consider
                guard :
                        r2.Skill2ConsiderDomainRestriction(),

        r3 : m1!Skill2Consider
                guard :
                        r3.Skill2ConsiderDomainRestriction(),

        r4 : m1!Skill2Consider
                guard :
                        r4.Skill2ConsiderDomainRestriction()
{
match: validatePatternallDifferent(r0,r1,r2,r3,r4)
onmatch
        {
        transformPattern(r0,r1,r2,r3,r4);
        }
}
```

Listing 1: Excerpt of the EPL pattern generated for constraint 3.

```
operation m1!Profile ProfileDomainRestriction(): Boolean
{
        return true;
}
operation m1!Skill2Consider
      Skill2ConsiderDomainRestriction(): Boolean
{

      return not self.maxLevel() and self.hasPrerequiste();
}
```

Listing 2: Domain restriction resource.

caught by a role. Listing 2 illustrates the way the constraint 2 is expressed as a domain restriction. The *maxLevel*() and *hasPrerequiste*() operations are defined in the context of the *Skill2Consider* metaclass (*Skill2ConsiderOperations.eol*). The added code is highlighted.

Listing 3 depicts an excerpt of the validation resource. The two operations are automatically generated with respect to the constraints type and names. We complete these operations with action blocks that express the specific validation logic for models matched by the pattern. The *allDifferent*() and *following-*

NotMatch() operations are predefined operations. Indeed, we defined a list of operations (e.g. *sort*(), *maxOccurs*()...) which are automatically added to the operations resource.

```
operation validatePatternallDifferent(r0 : ml!Profile,r1
      : ml!Skill2Consider,
        r2 : ml!Skill2Consider,r3 : ml!Skill2Consider,r4
      : ml!Skill2Consider) :Boolean
{
        return allDifferent(Sequence{r1,r2,r3,r4});

}
operation validatePatternfollowingNotMatch(r0 :
      ml!Profile,r1 : ml!Skill2Consider,
        r2 : ml!Skill2Consider,r3 : ml!Skill2Consider,r4
      : ml!Skill2Consider) :Boolean
{
        return followingNotMatch(Sequence{r1,r2,r3,r4});
}
```

Listing 3: Validation resource.

As for the transformation resource (see Listing 4), we complete it with actions to be applied to the match model in order to produce a valid objective scenario. A predefined launch configuration is provided to perform this task.

```
operation transformPattern(r0 : ml!Profile,r1 :
      ml!Skill2Consider,
        r2 : ml!Skill2Consider,r3 : ml!Skill2Consider,r4 :
      ml!Skill2Consider)
{
        var s=createS();
        var os=createOS();
        s.objectiveScenario=os;
        for(r in Sequence{r1,r2,r3,r4})
        {
                os.targetedSkills.add(createTS(r));
        }
}
operation createS() : m3!Scenario
{
        return new m3!Scenario;
}
operation createOS() : m3!ObjectiveScenario
{
        return new m3!ObjectiveScenario;
}
operation createTS(s : ml!Skill2Consider) : m3!TargetedSkill
{
        var ts= new m3!TargetedSkill;
        ts.difficultyLevel=s.currentLevel;
        ts.bxSkill=s.bxSkill;
        return ts;
}
```

Listing 4: Transformation resource.

Figure 10 presents the generated scenario. We can see that it satisfies constraints 1 and 2. However, the proposed skills do not satisfy constraint 3. In fact, no combination of the possible skills allows enforcing the constraint because the skill B3 is acquired at its highest level. For that, the transformation generates a scenario with respect to a less prioritized constraint (constraint 4 in our case). Indeed, the sequencing of the two generated EPL patterns (i.e. for enforcing constraints 3 and 4) is derived from the configuration model and automatically expressed by means of an ANT-based Epsilon workflow (c.f. Section 3.3).

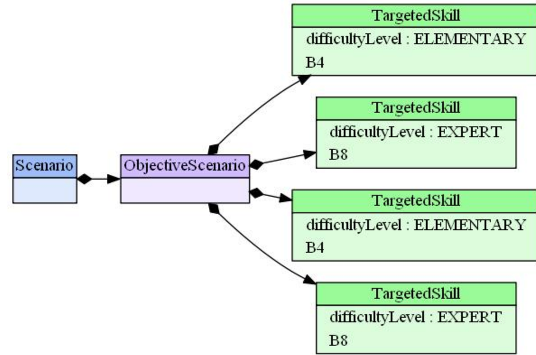

Figure 10: The generated objective scenario.

5 RELATED WORK

Our approach for constraint solving is based on expressing constraints to enforce on target models by means of source model elements. This proposal is inspired from graph transformations techniques where constraints on the involved graphs can be expressed through application conditions (Ehrig et al., 2004). Besides, the proposed transformation process (i.e. including the match and transformation steps) is similar to the application of graph transformations. For these latter, the source graph fragments concerned with the application of a transformation are first determined with respect to the LHS (Left Hand Side) graph. Then, the matched fragments are replaced with the structure of the RHS (Right Hand Side) graph.

The main difference is the way the pattern is defined. In fact, the pattern structure is separated from the constraints which allows expressing different and prioritized constraints for the same pattern. Besides, it is much easier to express complex constraints within our approach (e.g. textual syntax, feature navigation, predefined operations...). In contrast, for graph transformations the pattern is defined as one block (i.e. the LHS graph) and constraints are expressed like subgraphs.

Several proposals have addressed the problem of directly enforcing constraints on target models. Petter et al. (Petter et al., 2009) have proposed an implementation to extend the QVT-Relations language (OMG, 2008) with constraint solving capabilities. However, the proposal focuses on constraints related to attribute values and disregards global constraints.

Other related work address the automatic generation of models. In this case, models are not considered as targets of applying model transformations but are viewed as valid instances of constrained metamodels (Kleiner et al., 2010). Cabot et al. (Cabot et al., 2008) have proposed an approach where metamodels and

OCL constraints are translated into a CSP and a dedicated solver allows producing a valid instance. Based on similar principles, Ferdjoukh et al. (Ferdjoukh et al., 2013) have proposed an approach for model generation while dealing with performance.

In the limited scope of the presented use case, we have experimented the use of model generation techniques to perform the transformation scenario. The idea was to express the source models information, the way to construct the target model and the expert requirements, as OCL constraints. Hence, a model generation support (we chose Grimm (Ferdjoukh et al., 2013)) can be used to deal with the generation of the expected objective scenario. However, the tool failed because it does not support some essential OCL operations.

6 CONCLUSION

In this paper, we have presented a practical approach for constraint solving in model transformations. The base principle is to consider pattern matching problem as a high level specification of CSP. Furthermore, the pattern structure is decoupled from the validation constraints so as to allow associating multiple constraints to a same pattern and specifying shared transformation rules for all validation logics.

This proposal also includes a transformation infrastructure. This latter is generated in a semi-automatic manner and it provides support for pattern specification, match model search, and transformation into valid target models. A use case has been selected to illustrate these tasks.

We are currently exploring the definition of parameterized patterns essentially for avoiding the problem of fixed roles numbers. Hence, the same pattern definition can be used in various transformation scenarios even if involving slightly different match models. Besides, we intend to integrate this proposal in a co-design and validation framework for the presented serious game.

In this case, we have to deal with two issues. First, the regeneration of the transformation infrastructure must consider the extent of the variation expressed by the expert (e.g. adding a constraint must imply changing the validation resource without impacting the transformation and operations resources). Second, we found that even if several solutions are possible, the same scenario is always generated for different executions. This confuses the validation task and prevents the experts from considering several illustrations. Currently, we can set the transformation infrastructure in an iterative mode to allow the generation of all possible targets. Moreover, we are planning to implement a solution to address random generation.

REFERENCES

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

Cabot, J., Claris, R., Riera, D., et al. (2008). Verification of uml/ocl class diagrams using constraint programming. In *First International Conference on Software Testing Verification and Validation, ICST 2008.*, pages 73–80. IEEE.

Degueule, T., Combemale, B., Blouin, A., Barais, O., and Jézéquel, J.-M. (2015). Melange: A meta-language for modular and reusable development of dsls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 25–36. ACM.

Ehrig, H., Ehrig, K., Habel, A., and Pennemann, K.-H. (2004). Constraints and application conditions: From graphs to high-level structures. In *International Conference on Graph Transformation*, pages 287–303. Springer.

Ferdjoukh, A., Baert, A.-E., Chateau, A., Coletta, R., and Nebut, C. (2013). A csp approach for metamodel instantiation. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 1044–1051. IEEE.

Jouault, F. and Kurtev, I. (2005). Transforming models with atl. In *International Conference on Model Driven Engineering Languages and Systems*, pages 128–138. Springer.

Kleiner, M., Del Fabro, M. D., and Albert, P. (2010). Model search: Formalizing and automating constraint solving in mde platforms. In *European Conference on Modelling Foundations and Applications*, pages 173–188. Springer.

Kolovos, D., Rose, L., Garcia-Dominguez, A., and Paige, R. (2017). The epsilon book (2017).

Kolovos, D. S. and Paige, R. F. (2017). The epsilon pattern language. In *9th IEEE/ACM International Workshop on Modelling in Software Engineering, MiSE@ICSE 2017*, pages 54–60. IEEE.

Kolovos, D. S., Paige, R. F., and Polack, F. A. (2006). The epsilon object language (eol). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer.

Kolovos, D. S., Paige, R. F., and Polack, F. A. (2008). The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer.

Laforcade, P., Loiseau, E., and Kacem, R. (2018). A model-driven engineering process to support the adaptive generation of learning game scenarios. In *Proceedings of the 10th International Conference on Computer Supported Education*.

Larrosa, J. and Valiente, G. (2002). Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4):403–422.

Mens, T. and Gorp, P. V. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142.

OMG (2008). Meta object facility (mof) 2.0 query/view/-transformation specification.

OMG (2014). Object constraint language 2.4 specification.

Petter, A., Behring, A., and Mühlhäuser, M. (2009). Solving constraints in model transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 132–147. Springer.

Rudolf, M. (1998). Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *International Workshop on Theory and Application of Graph Transformations*, pages 238–251. Springer.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

Taentzer, G., Ermel, C., and Rudolf, M. (1999). The agg approach: Language and tool environment. *Handbook of graph grammars and computing by graph transformation*, 2:551–603.