2013 IEEE 25th International Conference on Tools with Artificial Intelligence

A CSP Approach for Metamodel Instantiation

Adel Ferdjoukh, Anne-Elisabeth Baert, Annie Chateau, Rémi Coletta and Clémentine Nebut
LIRMM, Université Montpellier 2 and CNRS,
Montpellier, France
{*ferdjoukh, baert, chateau, coletta, nebut*}*@lirmm.fr*

***Abstract*—This work is a contribution of Artificial Intelligence to Software Engineering. We present a comprehensive approach to metamodel instantiation using CSP. The generation of models which conform to a given metamodel is a crucial issue in Software Engineering, especially when it comes to produce a variate and large dataset of relevant models to test model transformations or to properly design new metamodels. We define an original constraint modeling of the problem of generating a model conform to a metamodel, also taking into account its additional OCL constraints. The generation process we describe appears to be quicker, more efficient and flexible than any other state-of-the-art approach.**

***Keywords*-CSP; Model Driven Engineering; Metamodel**

I. Introduction

Model generation ([1], [2]) is a problem stemming from Model-Driven Engineering (MDE). MDE is a recent paradigm that recommends the intensive use of structured models during all the lifecycle of a software, from the early requirements down to the testing phases ([3], [4]). The various models have different structures according to the different purposes they are designed for. Such a structure is defined by a model called a metamodel. The models can be handled by programs, which are called model transformations. While testing such model transformations or while designing a metamodel, i.e. a new structure for models, a model generation mechanism is of prime importance, to obtain test data or to check if the designed metamodel structures adequate models [5]. The main properties that are required for a proper model generation are: (i) scalability, which implies a reasonable generation time, (ii) validity, meaning that the generation process must take into account additional constraints on the metamodel, namely Object Constraint Language (OCL) constraints, (iii) flexibility, implying that it can easily be parameterized, depending on the purpose of the automated generation, (iv) diversity of the solutions.

Model generation have been studied from several angles, like grammar graphs ([6], [5]), constraint solving with the Alloy constraint solver ([7], [8]), random graphs ([9]), and a first approach by CSP modeling was proposed by Cabot et al. in [10], [11]. These approaches present interesting points of view, however none of them gather all the required properties expected for model generation. The contribution of the paper is a model generation mechanism based on CSP, that can fit all these properties. In this paper, we propose a modeling of a metamodel as a CSP. We show that this modeling is adequate to support additional constraints coming from OCL constraints completing the metamodel. We validate our modeling with experiments of model generation from several metamodels taken from the literature.

The rest of the paper is organized as follows. In Section II, we detail the concepts of model and metamodel, as well as OCL constraints. In Section III, we analyze the existing work on model generation to identify the improvable aspects. In Section IV, we present our original CSP modeling of the metamodel instantiation problem. In Section V, we detail our experiments performed on several datasets.

II. Context and preliminary definitions

This section presents the basic concepts and definitions concerning the objects we aim to model in Section IV.

A. Models and metamodels

A model is a representation of a system or part of it, with a given objective. A model is written using a modeling language, for example, the Unified Modeling Language (UML)[1]. A metamodel is the model of such a modeling language, it defines the concepts that can be used while designing a model. For example, the UML metamodel defines the concepts that can be used in a UML model: classes, attributes, associations, use cases, etc, and how the concepts are linked through relations.

A model is said to conform to its metamodel. Metamodels being models, they also have to conform to a metamodel, that is called a meta-metamodel. It is usually accepted that this meta-metamodel should be unique, and can define itself, so that the meta-metamodel conforms to itself. In this paper, we use `ecore` (from the EMF [12]) as meta-metamodel. Since models and metamodels are usually defined with object-oriented languages, a model (resp. a metamodel) can be seen as an instance of its metamodel (resp. meta-metamodel). The conformance relations are represented in Figure 1. A metamodel is often accompanied with a set of constraints that have to be respected by all the models. Those constraints are usually called well-formed rules, and expressed with the Object Constraint Language (OCL)[1].

[1]http://www.uml.org/

1082-3409/13 $31.00 © 2013 IEEE
DOI 10.1109/ICTAI.2013.156

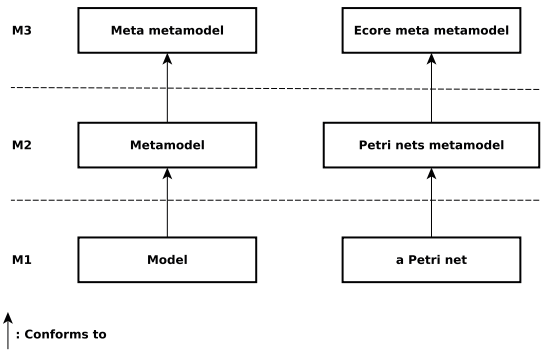

Figure 1. L.h.s: the general modeling hierarchy and conformance relations; R.h.s: example of the Petri net modeling hierarchy.

To illustrate those notions, we use the example of Petri nets. Figure 2 shows a metamodel for the Petri nets, that is taken and slightly adapted from reference [13]. This metamodel is drawn using the UML class diagram syntax. A Petri net has a name and is composed of several nodes and several arcs. There are two types of nodes: transitions and places. A place has a marking, that is an integer (precisely an `EInt` here, that is the `ecore` type for integers. Note that all the `ecore` types begin by the letter `'E'`). An arc has a weight, that is also an integer. An arc has one source and one target, that are nodes that have this arc respectively in its outgoing or ingoing arcs.

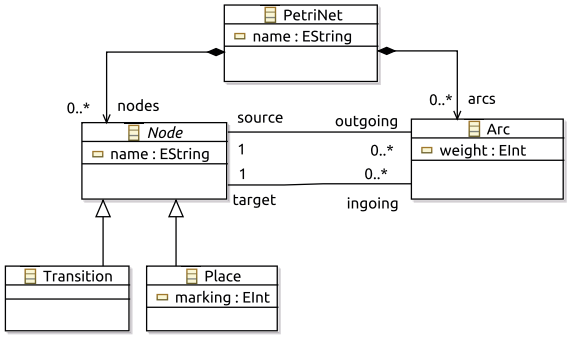

Figure 2. A metamodel for Petri nets, slightly modified from [13] .

Example 1 (OCL constraints for Petri nets): The Petri nets metamodel comes with the three following OCL constraints:

1) Distinct nodes must have distinct names:
```
context PetriNet inv :
self.nodes->forAll(n1 , n2 |
            n1 <> n2 implies n1.name <> n2.name)
```
2) An arc must not link two places or two transitions:
```
context Arc inv :
self.source.oclType( ) <> self.target.oclType()
```
3) The marking of a place must be positive:
```
context Place inv : self.marking >= 0
```

Let us focus on the simple Petri net model presented with a usual syntax in Figure 3. This model can also be seen as an instance of the Petri net metamodel: this is illustrated in Figure 4 using a UML instance diagram. In this diagram, we find two instances of the metaclass `Place` with respective names `place1` and `place2`, and respective marking value 2 and 3. We find an instance of the metaclass `Transition`, and two instances of the metaclass `Arc`, linked with the different places and transitions with links `source` and `target`. This model respects the three OCL constraints presented above.

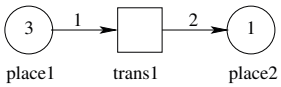

Figure 3. A model for a Petri net.

A model generation mechanism thus aims at generating instances of a given metamodel, such as the one presented in Figure 4. In the following subsection, we describe properties that a model generation should enforce.

B. Properties for model generation

Several properties are expected from a model generation mechanism.

- Scalability. The proposed approach must scale to large models and large metamodels. Indeed, large models are useful to test model transformations, for example to test their ability to scale.
- Validity. Only valid models must be generated. The generated models must conform to their metamodel, and respect the OCL constraints. Consequently, a model generation mechanism must be able to take into account the OCL constraints and a whole metamodel.

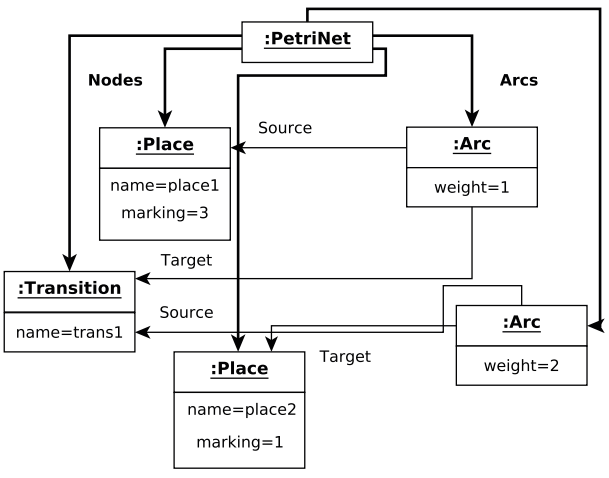

Figure 4. A model for a Petri net, seen as an instance of the Petri net metamodel of Figure 3.

- Flexibility. Since model generation can be used for different purposes such as model transformations testing and metamodel adjustment, model generation must easily be parameterized so as to generate small or large models, similar or dissimilar models, etc.
- Diversity and coverage. The generated models must be as representative as possible of the metamodel, and should well cover the metamodel while generating a large diversity of models.

III. Related work

Model generation being of prime important for model transformation testing and metamodel validation, it has been studied in the literature. In this section, we give an overview of the proposed approaches, and then go more into details for the approach proposed by Cabot *et al* [11], that is the nearest from the one we propose.

A. Several paradigms for model generation

Several approaches were proposed in the fields of Model Driven Engineering to propose model generation mechanisms, automatically instantiating a metamodel.

In [9], an approach based on random graph generation is exposed. To be more precise, random tree generation is used. The covering tree of the metamodel, formed by the containment references, is encoded into generating functions, that are later used in the generation process. The actual generation process is not deeply explained, however it seems that many ad-hoc generators have to be implemented. In this approach the OCL constraints are not taken into account (or only with ad hoc generators, such as generators for the inheritance relations, ensuring that there is no cycle in the relation), and the approach only generates skeletons of models (the covering tree) that have to be completed using another approach so as to obtain a complete model. Moreover, this approach is not flexible.

In [6], [5], the authors propose an approach based on grammar graph. The idea is to introduce elementary grammar rules encoding a metamodel, and applying them until the obtention of a conform model. Unfortunately, this approach does not deal properly with OCL constraints. The authors give hints to take part of the OCL constraints into accounts, but the whole OCL constraints can hardly be translated. Moreover, this approach is not easily flexible to introduce external constraints precising additional requirements on the generated models.

In [7], [8], the authors translate a metamodel and its OCL constraints into the Alloy language, and use the Alloy solver to obtain solutions. The mapping from a metamodel to Alloy is rather easy, since the two paradigms are not so far. However, the approach suffers from a scalability problem. The Alloy solver is dedicated to model checking and does not allow to quickly obtain solutions.

Approach	scalability	validity	diversity	flexibility
random graphs[9]	?	no	no	no
grammar graphs[6], [5]	yes	no	yes	no
Alloy [7], [8]	no	yes	no	medium
CSP [11]	no	yes	possible	yes

Table I
Comparison of the different approaches for model generation

Moreover, none of these approaches can benefit from the following facilities of CSP to achieve:

Diversity Thanks to recent works in solution counting [14], [15] CSP provide a way to generate a random set of solutions. Another alternative is to generate a solution far from an other one. In this case work on the distance constraint [16] may apply.

Flexibility: OCL constraints enforce hard constraints over the generated models, but do not prevent from generating dummy models, such as an UML model with only one class per package. But an expert is able to detect such dummy models. Recent works on constraint acquisition [17], [18] would be a way to learn extra constraints, in our example: *the number of package in a realistic UML model is half the number of classes*.

B. CSP for model generation

In [11], the authors propose an approach based on CSP, the metamodel as well as the OCL constraints are encoded into a CSP, and the ECLiPSe solver [19] is used to generate a solution (or show the absence of it). The approach was first proposed in the UML context [10]. Judging by the given results, this approach suffers from scalability problems. However, this problem does not mean that CSP is not suitable for the model generation problem. Indeed, analyzing the CSP modeling proposed in [11], one can notice several points that explain the limited performances. First of all, list variables are intensively used, when they are not necessary. Then disjunctive constraints are used (such as nth). Global constraints are never used is the modeling. Finally, the OCL is systematically and brutally introduced through a direct translation into prolog. In this paper, we show that a correct CSP modeling allows good performances.

C. Comparaison of the paradigms used in the literature

A comparison of the approaches presented above w.r.t. our four criteria is given in Table I. None of the approaches fits all the criteria. The last line concerns the use of CSP by Cabot et al. The main problem in this work concerns the scalability, that is due to problems in the CSP modeling. The properties of diversity and flexibility are not tackled by Cabot et al, however, we claim that a CSP based approach can benefit from several works in the literature in order to fit those two properties. Diversity and coverage can be achieved using a distance to put away the different solutions computed by the solver [16]. Flexibility can be enhanced using for example learning mechanism presented

in [18], [17] to learn implicit constraints establishing the characteristics of a relevant model, so as to generate only meaningful models. The approach presented in this paper presents all the properties to fit the four criteria.

IV. The CSP model

We propose here a modeling in which we consider that all features and references are propagated in all the sub-classes.

To take into account the notion of inheritance between the classes of a metamodel, we copy the features and the references from the super-classes into the sub-classes.

All along this section, we use the simple metamodel of Figure 5 in order to illustrate the different modeling steps from a metamodel to CSP.

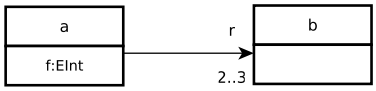

Figure 5. Metamodel containing two classes and one reference.

A. Classes modeling

1) Variables and domains: A class is modeled by two kinds of variables: variables representing the instances of the class and variables representing the features of each instance. We consider a given metamodel $\mathcal{M}$, with a set of n classes $\{c_1, \ldots, c_n\}$.

We denote by $Maxsize(c)$ (resp. $Minsize(c)$) the maximal (resp. minimal) number of instances of a class c. For each class c_k, and each $i \in \{1, \ldots, Maxsize(c_k)\}$, we denote by $Id_{c_k,i}$ the variable representing the i^{th} instance of the class c_k. We note $M_k = \sum_{j=1}^{k} Maxsize(c_j)$. The *domain* $D(Id_{c_k,i})$ of $Id_{c_k,i}$ is:

$$\begin{cases} \{M_{k-1}+1, \ldots, M_{k-1}+Minsize(c_k)\} & \text{if } i \leq Minsize(c_k), \\ \{0\} \cup \{M_{k-1}+Minsize(c_k)+1, \ldots, M_k\} & \text{otherwise.} \end{cases}$$

Note: Different class domains are disjoint, except for the value 0. This latter value means that the instance is not allocated. In addition, the root class only has one instance.

A class feature of simple type is modeled using a variable whose type is the domain: for each simple feature f of c and for each $i \in \{1, \ldots, Maxsize(c)\}$, we create a variable $F_{c,i,f}$ whose domain is given by:

$$D(F_{c,i,f}) = Type(f).$$

Enumerations are more complex features, that are modeled as follows: let $enum$ an l literals enumeration, f a feature of type $enum$. The domain of $F_{c,i,f}$ is given by:

$$D(F_{c,i,f}) = \{1, \ldots, l\}.$$

2) Constraints: Values affected to the instances of the different classes must be different. Only 0 can be affected to different variables because it indicates a not allocated instance. To answer this issue, we post the following Gcc [20] constraint.

$$\begin{array}{lll} Gcc(& [Id_{c_1,0}, \ldots, Id_{c_k,Maxsize(c_k)}], & \text{// variables} \\ & [0, \ldots, M_k], & \text{// values} \\ & [0, \ldots, 0], & \text{// lower bounds} \\ & [\sum_{j=1}^{k} Minsize(c_j), 1, \ldots, 1]). & \text{// upper bounds} \end{array} \quad (1)$$

Example 2: Let us illustrate our modeling on the metamodel in Figure 5 when the user asks for $[1-3]$ instances of class a and $[3-5]$ instances of class b. This kind of requirement on the number of instances for each class participates to the flexibility property. We fulfill this requirement by adapting the domain of the involved variables.

We create the following CSP variables:

- variables $\{Id_{a,1}, Id_{a,2}, Id_{a,3}\}$ represent the instances of class a. Their domains are: $D(Id_{a,1}) = \{1\}$, and $D(Id_{a,2}) = D(Id_{a,3}) = \{0, 2, 3\}$.
- variables $\{F_{a,1,f}, F_{a,2,f}, F_{a,3,f}\}$ represent feature f value for each instance of a. The domain of these variables is: $D(f) = [-50, 50]$
- variables $\{Id_{b,1}, Id_{b,2}, Id_{b,3}, Id_{b,4}, Id_{b,5}\}$ represent the instances of class b. The domains of these variables are: $D(Id_{b,1}) = D(Id_{b,2}) = D(Id_{b,3}) = \{4, 5, 6\}$ and $D(Id_{b,4}) = D(Id_{b,5}) = \{0, 7, 8\}$.

Please note that the domains of instances of distinct classes are disjoint, except for the value 0 which belongs to the domain of non-mandatory instances variables.

We also post the following Gcc constraint:

$$\begin{array}{l} Gcc([Id_{a,1}, \ldots, Id_{a,4}, Id_{b,1}, \ldots, Id_{b,5}], \\ [0, 1, \ldots, 8], [0, \ldots, 0], [8, 1, \ldots, 1]). \end{array} \quad (2)$$

This Gcc constraint stipulates that the value 0 can appear 8 times which is the total number of variables, whereas the other values $\{1, \ldots, 8\}$ can appear only once among all instances variables. All instances variables must have different values except for the value 0.

A solution to this CSP is, for instance:
$Id_{a,1} = 1, Id_{a,2} = 2, Id_{a,3} = 0$
$F_{a,1} = -10, F_{a,2} = 15, F_{a,3} = 0$
$Id_{b,1} = 6, Id_{b,2} = 5, Id_{b,3} = 7, Id_{b_4} = 8, Id_{b,5} = 0$

B. References modeling

The most interesting idea in our references modeling is considering them as pointers from class instances to others. While the modeling proposed by *Cabot et al.* in [10], [11]

treats a reference instance as a pair of variables (a, b) where a class instance a references a class instance b, we propose to model a reference using only one variable associated to a class instance a. It will take the value assigned to the variable of class instance b referenced by a.

Let c a metamodel class and $c.AllReferences$ the set of all references of c. Let $r \in c.AllReferences$ a reference of c. We denote by $LowerBound(r)$ (resp. $UpperBound(r)$) the lower (resp. upper) bound of r.

For each $i \in \{1, \ldots, Maxsize(c)\}$ and for each reference $r \in c.eAllReferences$, we create j variables $Ref_{i,j}^{c,r}$, where $j \in \{1, \ldots, UpperBound(r)\}$.

The set of all types of the reference r is denoted by:

$$S_{dst}(r) = \begin{array}{l} r.EReferenceType \cup \\ r.EReferenceType.getSubTypes(), \end{array}$$

where $getSubTypes()$ designates the sub-types set of a class and $EReferenceType$ returns reference destination class. The domain of the variable $Ref_{i,j}^{c,r}$, noted $D(Ref_{i,j}^{c,r})$, is given as follow:

$$\begin{array}{l} \cup_{c \in S_{dst}(r)}(D(Id_{c,1})), \text{ if } j < LowerBound(r), \\ \cup_{c \in S_{dst}(r)}(D(Id_{c,1})) \cup \{0\}, \text{otherwise.} \end{array}$$

It means that the $LowerBound(r)$ first variables must be allocated, thus, 0 does not belong to their domain. The other variables are optional, therefore their domain contains 0.

We also add the following constraints to the CSP:

- For each class c and r a reference of c, for each $i \leq Maxsize(c), j \leq UpperBound(r)$:

$$(Id_{c,i} = 0 \wedge 0 \in D(Ref_{i,j}^{c,r})) \leftrightarrow (Ref_{i,j}^{c,r} = 0).$$

  When a class instance is not allocated, no reference instance associated to it should be allocated.
- Let c_r the root class of a metamodel and $Refs(c_r)$ the set of its references. We define the following Gcc constraint, $\forall r \in Refs(c_r), j \in \{1, \ldots, UpperBound(r)\}$:

$$Gcc([Ref_{1,j}^{c,r}], [vals], [0, \ldots, 0], [h, 1, \ldots, 1]),$$

  where $vals = \cup(D(Ref_{1,j}^{c_r,r}))$ is the set of values which can be affected to the $Ref_{1,j}^{c_r,r}$ variables and $h = \sum_{r \in Refs(c_r)} UpperBound(r)$ the number of created reference variables.

Example 3: To model the reference r linking the two classes a and b in the metamodel in Figure 5 we create the following CSP variables:

- For each variable $Id_{a,i}$ modeling the instances of class a, we create 3 variables $\{Ref_{i,1}^{a,r}, Ref_{i,2}^{a,r}, Ref_{i,3}^{a,r}\}$, with $D(Ref_{i,1}^{a,r}) = D(Ref_{i,2}^{a,r}) = \{4, 5, 6, 7, 8\}$ and $D(Ref_{i,3}^{a,r}) = \{0, 4, 5, 6, 7, 8\}$. Indeed, the third variable is optional, then its domain should contain 0 in the case this variable is not used.

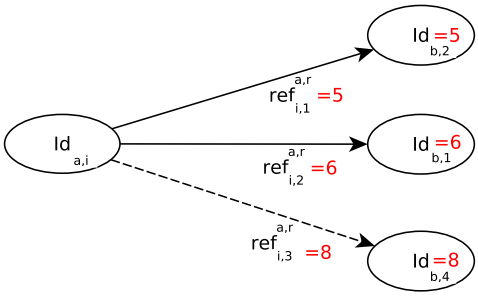

Figure 6. An example of value affectation for variables modeling the reference.

These variables can be seen as pointers from class a instances variables $Id_{a,i}$ to class b instances variables $Id_{b,j}$.

An example of the affectation of these variables is shown in Figure 6. When a variable $Ref_{i,j}^{a,r}$ is equal to a value v, we consider that the instance of class a represented by the variable $Id_{a,i}$ references the instance of class b represented by the variable which is affected v. As it is visible in Figure 6, class a instance references 3 instances of class b. The dashed line in the figure indicates that this variable is optional which means that its domain contains 0.

C. CSP model for OCL constraints

Object Constraint Language (OCL) is a formal language, based on predicates logic. It allows the expression of constraints on metamodels, as illustrated in Example 1.

In this paper, all the possible constructions of OCL are not treated but only the most widespread. However, we think that the other constructions are also feasible in our CSP model.

In the case of OCL constraints about a class feature, we propose to limit the domain of the variables representing these features. When we model a class feature, we create a domain containing positive and negative values. The CSP model of this first type of OCL constraints is only the limitation of the domain by the application of a boolean expression on each value in the domain.

For example, we define an OCL constraint on the class `Place` of Petri nets metamodel in Figure 2. This constraint stipulates that the value of feature marking should always be positive. To model this OCL constraint we limit the domain $[-50, 50]$ created at the modeling of the feature marking and subtract the negative values. The new domain for feature marking becomes $[0, 50]$. This model does not create any new CSP variable or constraint so it remains efficient.

The reference navigation mechanism of OCL is used in nearly all constraints, it allows a constraint to browse the metamodels elements through relations. To model this construction we have to verify an equality between a reference CSP variable (the pointer) and a class instance variable (the

pointed). The treatment of the OCL constraint is applied to all the couples verifying this equality. The treatment of references in our model as pointers allows to efficiently treat reference navigation in OCL constraints.

There are OCL constraints about collections which imply the creation of a binary constraints sets. To remain efficient, our model transforms these binary constraints into a global constraint *Alldiff* [21].

The CSP model we propose for OCL constraints takes into account the particularity of each construction of OCL language. This allows our solution to remain efficient. Conversely, the existing solution of *Cabot and al.* creates the same CSP model for each OCL constraint.

D. Comparing to Cabot et al. CSP model

We can observe some lacks in *Cabot and al.* CSP model proposed in [10]. Our solution corrects the failures and brings improvements to the CSP model in order to get an efficient generation process. The main improvements are the following:

- The previous CSP model uses list variables to model the instances of each class to distinguish classes one from the others. This slows constraints propagation. We use a set of instances variables for each class. To distinguish each instance and its class, we create contiguous interval domains.
- The major part of the constraints between variables in our model are global constraints. These constraints spread more easily and reduce considerably the number of constraints created. *Cabot and al.* CSP model does not use any global constraint.
- The preceding CSP model involves lots of disjunctive constraints and constraints which achieve only few propagation, such as the Element constraint (*nth*, which returns the n^{th} element of a list). Disjunctive constraints must be avoided in an efficient CSP model. In addition, separating the lists into simple variables in our CSP model implies the suppression of the *nth* constraint.
- Another interesting idea of our CSP model is to consider a reference between two classes as a pointer from instances of the source class to instances of the target class. This particularity reduces by half the number of variables representing references between classes comparing to the existing solution. It is also a good base to treat the OCL constraints of a metamodel.
- We propose an OCL constraints modeling as accurate as possible. It takes into account the particularities of each construction in OCL language. Indeed, we only limit the variable domain when an OCL constraint is about a class feature which is one of the most widespread OCL constraints construction and we create a global constraint when the OCL model creates a binary constraints set. Conversely, *Cabot and al.* model creates the same number of CSP constraints for all OCL constraints without looking to their characteristics.

V. Experiments

We use the EMF framework tools to develop a test program which takes a metamodel in entry and generates models conforming to it. The metamodel is transformed to a CSP problem. We write a CSP instance in the XCSP format. The *Abscon* solver [22] is used to solve CSP and returns solutions. The metamodel is instantiated according to the values returned by the solver and a valid model is generated.

We use the following metamodels for our experiments:

1) Petri nets metamodel (Figure 2);
2) Entities and Relationships metamodel (Figure 7);
3) B language specification metamodel ([23]);
4) Sad software architecture metamodel ([24]).

Their numbers of classes are presented in Table II.

The main purposes of our experiments are:

1) Comparing our solution to *Cabot and al. EMFtoCSP* tool and show that our CSP model is more efficient than the existing one. For that, we use the metamodel Entities and Relationships shown in Figure 7.
2) Verifying that our solution can generate models containing a high number of class instances and conforming to metamodels containing substantial number of classes. Thus we also use the metamodels (3) and (4), which are significatively large metamodels.

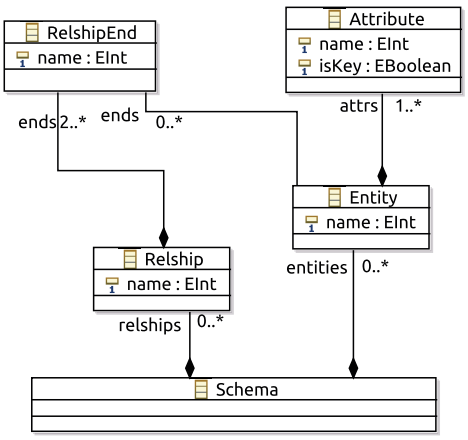

Figure 7. A metamodel for Entities and Relationships.

A. Comparing Our solution with Cabot and al. tool

The results in Figures 8 are obtained after the experiments carried on the Entities and Relationships metamodel (Figure 7). Resolution durations are given in CPU time. *Cabot and al.* experiments results are extracted from [11]. The curves

in Figure 8 show that the resolution with our solution is eight times faster than *Cabot and al.* tool [1].

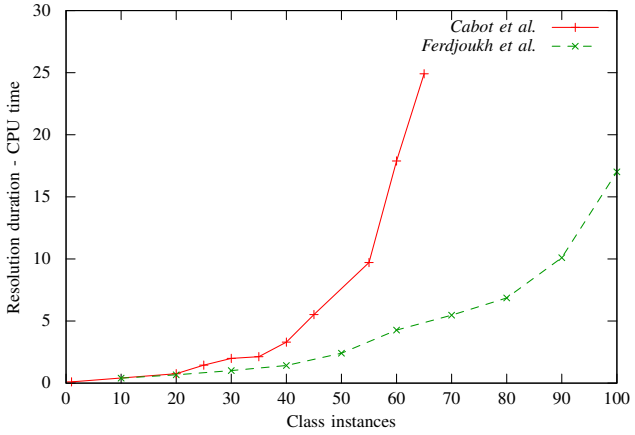

Figure 8. Comparison between our tool and *Cabot and al.*.

This performance gap is the result of the following improvements of our solution:

- The use of sets of variables to replace the lists used by *Cabot and al.*. Variables of type list are difficult to manipulate and they slow constraints propagation.
- Absence of global constraints in Cabot and al. CSP model. For example, Alldiff global constraint could replace the binary difference constraints introduced by their CSP model. Conversely, our model uses global constraints, for example the Global cardinality constraint (Gcc) which limits the number of appearance of a value in a set of variables.
- Representing class references as pointers from the source class to the destination class reduces by half the variables modeling the references comparing to *Cabot and al.* solution and also facilitates the navigation operation in OCL constraints.
- A more detailed modeling for OCL constraints of a metamodel increases the efficiency of our solution. Indeed, the CSP model of OCL constraints does not create any CSP variable. The model of some OCL constraints does not create CSP constraints because it consists in reducing variables domains.

B. Scaling of the solution

To measure efficiency of our solution relatively to the size of datasets, we carry experiments on four metamodels with different numbers of classes. The Table II shows the number of classes for each metamodel. The results of these experiments are shown in Figure 9 in which we make the global number of instances in generated models vary. We notice that our solution is able to generate models conform to metamodels containing a small number of classes and also for metamodels containing a substantial number of classes. The resolution time remains reasonable in all cases.

[1]The experiments have been driven on, if not the same, a slightly similar computer considering CPU and memory.

Metamodel	#Classes
Petri nets	4
Entities and Relationships	5
B language	34
Sad	40

Table II
NUMBER OF CLASSES OF THE DIFFERENT METAMODELS USED IN EXPERIMENTS.

The curves in Figure 9 show that the resolution time is reasonable also for metamodels containing a substantial number of classes. These good results are due to the following reasons:

- There are not many constraints in the CSP model we propose. When the number of CSP variables increases, the number of constraints stays the same so the efficiency is not affected.
- Global constraints like the Global cardinality constraint (Gcc) which covers a lot of variables are used, their propagation is easier than other constraints.

However, the results show also that, for a large number of instances generated, our modeling is less efficient on metamodels containing a smaller number of classes such as Petri nets (4 classes), than on metamodels containing a larger number of classes such as B language (34 classes). This can be explained by the quick increase of variables modeling the references.

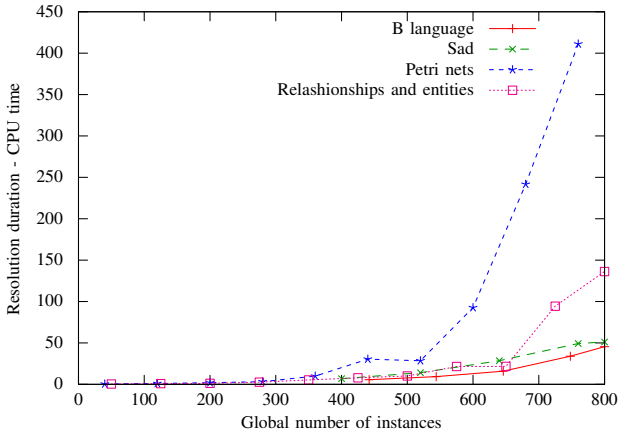

Figure 9. Comparing all metamodels experiments results.

VI. CONCLUSION

The model generation problem is a very challenging application for the Software Engineering community. The previous approaches lack of either efficiency or flexibility,

which are required to extend the generation process in a useful way. Due to a too basic modeling, even the previous CSP approach presents such drawbacks. However, our modeling based on a more careful treatment of the problem, saving a significant amount of variables and constraints, seems really promising. We achieve the generation of models with OCL constraints, even for large instances, in a very reasonable time, compared to the previous methods. Moreover, we expect that this issue may be improvable, by a careful study of the symmetries in the modeling of references.

Future works will focus on this aspect, as well as on the diversity of the solutions. Indeed, when we generate several solutions for the same metamodel, it is more interesting to get them homogeneously distributed in solution space. It means that we have to generate distant models (thus distant graphs), for example using the work presented in [16].

The second part of the future works is to generate relevant models. This is an important characteristic to practice model transformation tests. Our objective is here to learn implicit constraints through constraints acquisition mechanisms such as the ones presented in [17], [18].

ACKNOWLEDGMENT

The authors would like to thank Christophe Lecoutre and Vincent Perradin for their help to use Abscon Solver [22].

REFERENCES

[1] B. Baudry, T. Dinh Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon, "Model Transformation Testing Challenges," in *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing*, 2006.

[2] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Communications of the ACM*, vol. 53, no. 6, 2010.

[3] T. Stahl, M. Völter, and K. Czarnecki, *Model-driven software development: technology, engineering, management*, 2006.

[4] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, ser. Synthesis Lectures on Software Engineering, 2012.

[5] K. Ehrig, J. Küster, and G. Taentzer, "Generating instance models from meta models," *Software and Systems Modeling*, 2009.

[6] K. Ehrig, J. M. Kister, G. Taentzer, and J. Winkelmann, "Generating instance models from meta models." in *FMOODS*, 2006, pp. 156–170.

[7] S. Sen, B. Baudry, and J.-M. Mottu, "On combining multi-formalism knowledge to select models for model transformation testing," in *ICST*, 2008.

[8] S. Sen, B. Baudry, and J.-M. Mottu, "Automatic model generation strategies for model transformation testing," in *International Conference on Model Transformation*, 2009.

[9] A. Mougenot, A. Darrasse, X. Blanc, and M. Soria, "Uniform random generation of huge metamodel instances," in *5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA)*, 2009, pp. 130–145.

[10] J. Cabot, R. Clarisó, and D. Riera, "Verification of uml/ocl class diagrams using constraint programming," in *Proc. of the Software Testing Verification and Validation Workshop*. IEEE Computer Society, 2008, pp. 73–80.

[11] C. A. González Pérez, F. Buettner, R. Clarisó, and J. Cabot, "EMFtoCSP: A Tool for the Lightweight Verification of EMF Models," in *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*, 2012.

[12] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[13] J. J. Cadavid, "Assisting precise Metamodeling," Ph.D. dissertation, Université de Rennes 1, 2012.

[14] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting: A new strategy for obtaining good bounds," in *AAAI*, 2006, pp. 54–61.

[15] N. Creignou and H. Daude, "Generalized satisfiability problems: minimal elements and phase transitions," *TCS*, vol. 302, no. 1-3, pp. 417 – 430, 2003.

[16] E. Hebrard, B. O'Sullivan, and T. Walsh, "Distance constraints in constraint satisfaction," in *IJCAI*, 2007, pp. 106–111.

[17] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.-G. Quimper, and T. Walsh, "Constraint Acquisition via Partial Queries," in *IJCAI*, 2013, p. 7.

[18] N. Beldiceanu and H. Simonis, "A model seeker: Extracting global constraint models from positive examples," in *CP*, M. Milano, Ed., 2012, pp. 141–157.

[19] K. R. Apt and M. Wallace, *Constraint logic programming using Eclipse*. Cambridge University Press, 2007.

[20] J.-C. Régin, "Arc consistency for global cardinality constraints with costs," in *CP*, 1999, pp. 390–404.

[21] J.-C. Régin, "A filtering algorithm for constraints of difference in csps," in *AAAI*, 1994, pp. 362–367.

[22] S. Merchez, C. Lecoutre, and F. Boussemart, "Abscon: A prototype to solve csps with abstraction," in *CP*, 2001, pp. 730–744.

[23] J. Cadavid, "B language metamodel in ecore format + ocl well-formedness rules," INRIA, 2012.

[24] J. Cadavid, "Sad3 metamodel + ocl well-formedness rules," INRIA, 2012.