A generic approach to model generation operations

Mathias Kleiner[a], Marcos Didonet Del Fabro[b]

[a]*LSIS, Arts et Mtiers, Aix-en-Provence, France*
[b]*C3SL Labs, Informatics department, Federal University of Parana, Curitiba, Brazil*

Abstract

Model generation operations are important artifacts in MDE applications. These approaches can be used for model verification, model finding, and others. In many scenarios, model transformations can as well be represented by a model generation operation. This often comes with the advantage of being bidirectional and supporting increments. However, most part of model generation approaches do not target several operation kinds, but narrower scenarios by mapping the generation problem into solver specific problems. They are efficient, but often don't have a supporting framework. In this paper, we present an approach and framework that allows to specify and to execute model operations that can be represented in terms of model generation operations. We first introduce a model search layer that can be used with different solvers. We illustrate this layer with a driving example implemented using Alloy/SAT solver. On top of this, we introduce a transformation layer, which specification are translated into the model search layer, independently from any solver. The solution is natively bidirectional, incremental and it is not restricted to one-and-one scenarios. The approach is illustrated by two use cases and with 3 different scenarios, backed by a full, extensible and free implementation.

Keywords: Model Search, Model Transformations, Alloy

1. Introduction

In Model-Driven Engineering (MDE) approaches, studied or developed systems are captured through a set of models representing different structural and behavioural points of view. A model must comply to constraints which may be either generic rules that apply to any models of its kind (the language syntax and semantics), or system-specific considerations that stem from the user objectives. Therefore one kind of model operation is the ability to (semi)-automatically generate or complete a given partial (possibly empty) model. This operation, here called model generation, has different uses: model verification, language testing, use cases generation or user assistance in defining the system. Given the graph-like structure and mostly discrete properties of meta-languages, existing approaches to model generation usually rely on combinatorial techniques such as constraint programming solvers. The process thus consists in mapping the model generation problem to a solver-specific problem definition where resolution is achieved, and then mapping the solution(s) (if any) back to the modeling world. This approach is used for instance in [17, 30, 37]. The main drawbacks are the limitations of the chosen solver. Indeed, the nature of meta-languages yields hard combinatorial problems that may require solvers to deal with solutions of *a priori* unknown size, a mix of discrete and continuous variables, or complex strings manipulation. Therefore different problems might require different solvers, and often need to be simplified in order to be turned into viable specifications.

Additionally, the different models of a system are usually related, meaning that some, if not all, of a given model elements can be deduced from the others. This yields a second kind of operation, here called model transformation, where the goal is to obtain a set of (target) models from a set of (source) models. A first set of approaches that rely on rules and pattern matching [11] have been successful for a large number of use cases. However, the system development process is rarely linear, meaning that existing models may be modified and should still be kept consistent. These use cases have outlined the unidirectional and non-incremental limitations of most of these approaches, leading to studies in novel transformation techniques having bidirectional and incremental properties [21]. Some of these approaches, such as JTL [13], MOMoT [15] or Echo [34], propose to represente model transformations in terms of a model generation problem. However, the support for multiple kinds of generation operations could be improved.

In this article we present a generic approach and framework to specify and to execute model operations that can be represented in terms of model generation operations. This article re-founds and extends work presented in previous conference papers [30, 31]. The approach has the following main highlights.

Model search: we present a layer called *model search (MS)*, which handles the model generation process. This layer is broke down into solver-specific and solver-

Email addresses: `mathias.kleiner@ensam.eu` (Mathias Kleiner), `marcos.ddf@inf.ufpr.br` (Marcos Didonet Del Fabro) independent components. The solver-specific ones provide the extraction/injection of the input and output models into/from the solver format. The solver independent parts are the remaining components. We provide a completely rewritten open source implementation based on the Eclipse Modeling Framework (EMF) [11], where the executed chain can be easily adapted, improving considerably its applicability. We also provide a fully operational example implementation of the generic approach that targets the Alloy/SAT solver [21].

Multi-level transformations: we provide a generic bridge between the modeling problem and solver technical spaces, through the implementation of reflexive model transformation, called *multi-level transformations*. The transformations are implemented using only the metametamodels elements. They discover the metamodel and model elements during execution time. This enables having one single transformation for any input metamodels and their corresponding models, and without relying on an unification format. The same is valid for the output models.

Model transformations as search (TAS): we present a TAS layer that is independent from the underlying solver, conceptually and practically. This means that a model transformation specification is defined using only modeling components. These are transformed into a model search problem and solutions are then mapped back to the resulting models. In addition, the approach is not restricted to one-to-one transformation scenarios, it is multi-directional and incremental. We have developed as well a set of components to develop model transformations and to interact with the model search layer.

Unified formalism: finally, we revisited and provided an integrated conceptual view from both model and transformation as search.

Plan of the article. Section 2 provides the theoretical background. In Section 3, we formally define the model search layer, theoretically and with a practical guiding example. In Section 4 we describe the transformation and synchronization layer. We provide experimental results and analyse the strengths and drawbacks of the approach with additional comments on two examples from the literature and the industry. Section 5 presents the related work. Finally, we conclude in Section 6.

2. Context

2.1. Brief introduction to modeling and model transformation

Model Driven Engineering (MDE) considers models, through multiple abstract representation levels, as a unifying concept. The central concepts used in such approaches are terminal model, metamodel, and metametamodel. A terminal model is a representation of a system. It captures some characteristics of the system and provides knowledge about it. MDE tools act on models expressed in precise modeling languages. The abstract syntax of a modeling language, when expressed as a model, is called a metamodel. The relation between a model and the metamodel of its language is called conformance. Metamodels are in turn expressed in a modeling language for which conceptual foundations are captured in an auto-descriptive model called metametamodel. There are multiple model definitions in the literature (see [32] for a deep study), we refine in this article the ones introduced in [24] since simple graph-based definitions will prove useful in our context.

Definition 1 (model). *A model M is a triple $< G, \omega, \mu >$ where:*

- *G is a directed labelled multigraph,*
- *ω (called the reference model of M) is either another model or M itself (i.e., self-reference)*
- *μ is a function associating nodes and edges of G to nodes of G_ω (the graph associated to its reference model ω)*

Definition 2 (conformance). *The relation between a model and its reference model is called conformance and denoted $conformsTo$ or $c2$.*

Definition 3 (metametamodel). *A metametamodel is a model that is its own reference model (i.e., it $conformsTo$ itself).*

Definition 4 (metamodel). *A metamodel is a model such that its reference model is a metametamodel.*

Definition 5 (terminal model). *A terminal model is a model such that its reference model is a metamodel.*

Although the presented work may be adapted to other metalanguages, we will assume in the following the use of ECORE (an implementation of OMG's EMOF) as the metametamodel [12], since it is supported by a wide set of modeling tools. The main way to automate MDE is by executing operations on models. For instance, the production of a model Mb from a model Ma by a transformation Mt is called a model transformation. The OMG's Query View Transformation (QVT) [40] defines a set of useful model operations languages. In particular, it defines a language called QVT-operational which is restricted to unidirectional transformations scenarios, and a language called QVT-relational which can be used for bidirectional and synchronization scenarios.

2.2. Constrained metamodels

The notion of constraints is closely coupled to MDE. Engineers have been using constraints to complete the definition of metamodels for a long time, as it can be found in implementations combining UML/OCL (e.g., [1]). Constraints can be, for instance, checked against one given model in order to validate it. In our approach we will always consider metamodels with potential constraints attached. We first formally define the combination:

Definition 6 (constrained metamodel). *A constrained metamodel CMM is a pair $< MM, C >$ where MM is a metamodel and C is a set (a conjunction) of predicates over elements of the graph G associated to MM. We will consider an oracle that, given a model M, returns true (noted $M \in C(MM)$ where $C(MM)$ is the set of all valid models) iff M satisfies all predicates from C.*

The conformance relation between a model and its reference is then naturally extended to constrained metamodels.

Definition 7 (constrained conformance). *A model M* conformsTo *a constrained metamodel CMM iff it $conformsTo$ MM and $M \in C(MM)$.*

Many languages can be used to define predicates (i.e., constraints) with different levels of expressiveness. In this article, we will assume the use of OCL, though the presented work may be adapted to other constraint languages. Indeed, OCL is widespread, well integrated in modeling technologies, and expressive (it supports operators on basic datatypes, sets and relations as well as universal/existential quantifiers and various iterators).

2.3. Brief introduction to model finding

We call *model finding* the problem of finding and exhibiting a model (in its broad mathematical acceptance) from a given definition. Computational techniques for such problems is a vast area of theoretical and applied research and relates to various types of decision, satisfaction and optimization problems. Obviously, finding a model (in its MDE acceptance) that complies to a constrained metamodel is a model finding problem, in which the *search space* is implicitly defined by the set of potential well-formed models. Although this work does not assume any particular model finding technique, focus will be put on constraint programming (CP), the usual approach in modeling environments. CP is a declarative programming technique to solve combinatorial (usually NP-hard) problems. A constraint, in its wider sense, is a predicate on elements (represented by variables). A CP problem is thus defined by a set of elements and a set of constraints. The objective of a CP solver is to find an assignment (i.e., a set of values for the variables) that satisfies all the constraints. There are several CP formalisms and techniques [23] which differ by their expressiveness, the abstractness of the language and the solving algorithms. For instance, the SAT (boolean SATisfiability problem) formalism. A SAT problem is to decide if, for a given boolean formula, each boolean variable can be given an assignment such that the formula evaluates to true. SAT is known as being a NP-complete problem [8], and as such any CP problem can be reduced into SAT

Since SAT is a low-level formalism, manipulating only boolean variables, higher-level languages have been proposed to ease real problems specifications. One of those is Alloy [22], an expressive relational language that uses a built-in compiler (KodKod) to produce SAT problems. The Alloy tool offers to solve using several underlying SAT engines and translates solutions back to its relational idiom.

3. Model search

We consider the operation that aims at generating a complete and valid model of a constrained meta-model, starting with an incomplete (possibly empty) model. We first propose a formal model-based definition of such a task as a first-class model operation called *model search*. We then describe an example process as a generic pattern for solver-specific implementations. Finally, we describe a detailed example implementation using Alloy/SAT together with experiments.

3.1. Model search definition

In order to formally define model search, let us first define a set of notions that relate to constrained metamodels.

Relaxed metamodels and partial models.

Definition 8 (relaxed metamodel). *Let $CMM =< MM, C >$ (with $MM =< G, \omega, \mu >$) be a constrained metamodel. $CMM_r =< MM_r, C_r >$ (with $MM_r =< G_r, \omega, \mu >$) is a relaxed metamodel of CMM (noted $CMM_r \in Rx(CMM)$) if and only if $G_{MM_r} \subseteq G_{MM}$ and $C_r \subseteq C$.*

In other words, a relaxed metamodel is a less constrained (and possibly smaller) metamodel. A simple one can be straightforwardly obtained by the removal of all constraints: structural (making references and attributes optional) and external (removing predicates). Computing such a relaxed metamodel, a simple operation, is called *relaxation* in the following. In many frameworks, including ECORE-based ones, the relaxed metamodel does not need to be an additional concrete artifact, since the implementation is flexible enough to support it.

Definition 9 (partial model, p-conformsTo). *Let $CMM =< MM, C >$ be a constrained metamodel and M_r a model. M_r p-conformsTo CMM iff it conforms to a metamodel CMM_r such that CMM_r is a relaxed metamodel of CMM ($CMM_r \in Rx(CMM)$). M_r is called a* partial *model of CMM.*

Informally, a partial model is simply understood as being an incomplete or faulty model.

Model search.

Definition 10 (model search). *Let $CMM =< MM, C >$ be a constrained metamodel, and $M_r =< G_r, MM_r, \mu_r >$ a partial model of CMM. Model search is the operation of finding a (finite) model*

$M_s =< G_s, MM, \mu_s >$ such that $G_r \subseteq G_s$, $\mu_r \subseteq \mu_s$ (embedding i.e., $\forall x \in Gr, \mu_s(x) = \mu_r(x)$), and M_s $conformsTo$ CMM.

Informally, model search extends a partial model M_r into a model M_s conforming to its constrained metamodel CMM (or generates one when no M_r is given). In the following, M_r is called the *request* model, and M_s the *solution* model. The restriction that G_r is included in G_s could be removed if the solver supports removal of elements, or this could be circumvented by re-generating a complete new model, without the deleted elements.

This operation is illustrated in Figure 1. In other words, we consider model search as a operation where the request (metamodel and model) is an instance of a non-deterministic (often combinatorial) problem and the solution model is one of the results (if any exists). From the solver point of view, the request metamodel acts as the *problem definition* whereas the request model is a given *partial assignment*.

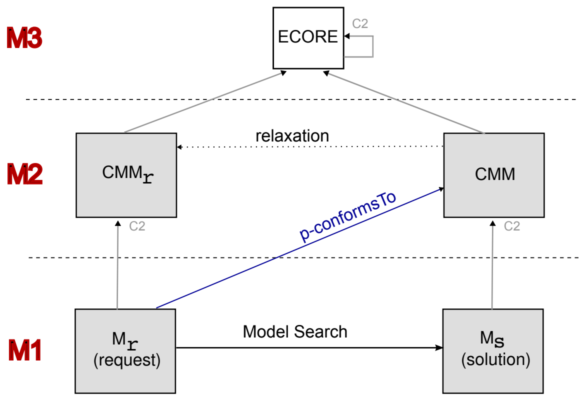


Figure 1: the model search operation

3.2. Model search process

We provide below an example generic process, independently of any solver, to explain the usual steps involved when implementing model search in a modeling environment. This software chain is illustrated in Figure 2, where dark gray squares are solver-specific parts. It is composed of 5 main tasks.

1) Problem definition generation: this task, illustrated by the $CMM2SP$ transformation, expresses the constrained metamodel as a solver problem definition. However, the $CMM2SP$ arrow is a simplified view of the operation, since there are actually two source models as input to the transformation. Figure 3 shows the actual transformation and its simplified view. The metamodel MM contains the structural constraints, which may be expressed, for instance, by ECORE. However, typical model search applications require more complex domain constraints (e.g., to set up a maximum cardinality value for an attribute). These domain constraints are expressed in the constraint model C, which can, for instance, conform to the OCL metamodel. The constraints refer to the elements of MM. Thus, a combination of ECORE+OCL could be one pair of input models.

The difficulty of expressing a constrained metamodel as a solver problem is highly dependent on the abstraction level and the basic elements offered by the target solver. Some implementation issues will be discussed in Subsection 3.4.

2) Partial assignment generation: this task is illustrated by the $M2SP$ transformation. It takes the request M_r as input and generates the corresponding partial assignment for the solver. Here the main difficulty is that the input metamodel MM is domain dependent, which means it may be different according to the search problem being considered. Since many transformation languages consider the input metamodel as un-changeable, it would imply writing a different transformation $M2SP$ for every considered metamodel MM. Clearly, this is undesirable. We propose a solution using **multi-level transformations**. A *multi-level* transformation is a model transformation that takes as input the domain model M and also the domain metamodel MM and that produces as output the solver partial assignment. This transformation is implemented using reflection. More detailed explanations on this multi-level transformation are given together with an implementation in Section 3.4.

It is important to note that many solvers do not separate the problem definition and the partial assignment: they are usually expressed using the same language/code. For that reason, both share the same "solver problem" metamodel. When this is not the case, the process is easily adapted by separating the partial assignment metamodel from the problem definition metamodel.

3) Engine program extraction: this task extracts the solver problem model into its parsable or executable format. Any classic model-to-text or model-to-code modeling technologies can be used here.

4) Solver execution: the generated solver file/program is executed in order to obtain solutions. When the search succeeds (i.e., there is at least one solution), we obtain a solution in the solver export format. The most common are XML or grammar-based text files.

5) Solution injection: this last task converts the solution(s) produced by the solver as model(s) of the original search metamodel MM. It is natural to decompose the operation into two sub-tasks: injecting the solution into the modeling environment based on the solver output format, then transforming to a model conforming to the original search metamodel MM. We have considered in this example process that the engine generates an XML file. Therefore we first do a straightforward injection of the XML solution in the modeling environment. If the solver rather produces grammar-based files, this can be

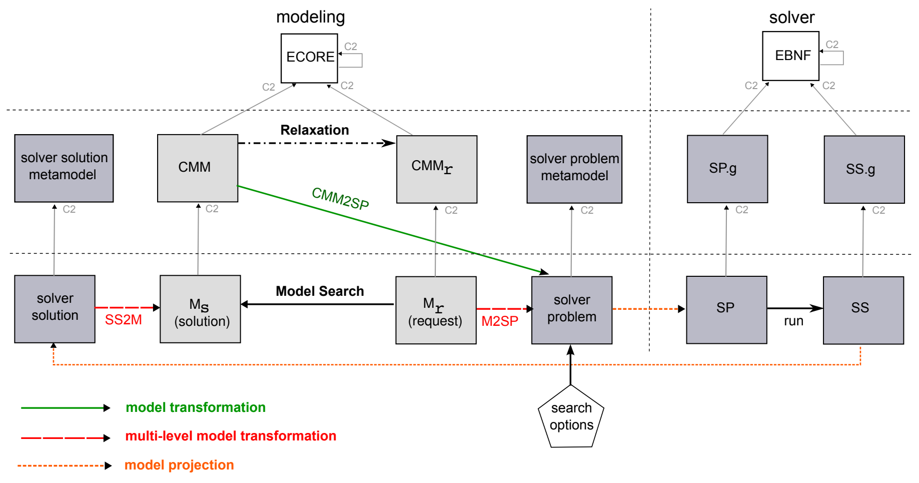


Figure 2: Model search example implementation process in a modeling environment

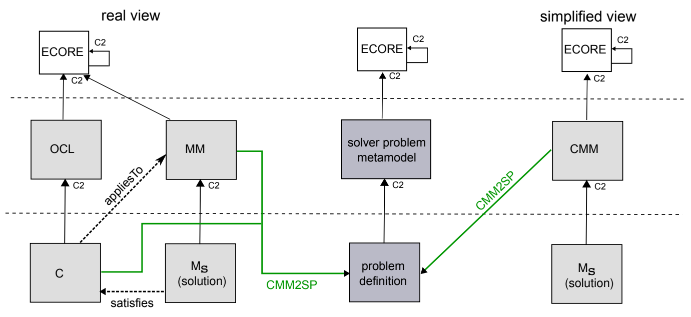


Figure 3: Generation of the problem definition

replaced by a classic text-to-model parser-based injection. Then we transform the output to a model conforming to MM ($M2SS$ in Figure 2). For the same reasons as the $M2SP$ transformation, $SS2M$ is a multi-level transformation: it takes MM as additional input and generates a model M_s. Again, more detailed explanations on this unusual multi-level transformation will be given in the example implementation.

3.3. Implementation of the generic part of the process

This part of the implementation regroups all the model search software parts (UI and API) that are solver-independent. This is of primary importance so that other model operations, presented in Section 4, can be defined independently from any solver-specific implementation. The implementation is distributed under the EPL license as a set of Eclipse plugins that are available for download at the MOS GitLab[1]. It is completely modular and extensible: the alternative solver-specific implementations are discovered through Eclipse's extension point mechanism. The plugins are divided in 4 main components:

1. *Launch configuration*: creates a launch configuration to set up the running parameters, which are the input and output models, metamodels and constraints.
2. *SolverChain*: it is the main API, which provides a set of classes and extension points for all the transformations of the chain, which need to be executed in the chain order. When the solvers need additional specific parameters for execution, they can be forwarded through a property/value list, called *model search options*, and then handled by the transformation.
3. *Solution explorer*: many existing solvers may produce more than one solution as output after the execution of the transformation chain, in other words, enabling several executions of the 4th task (*solver execution*). This component targets this kind of feature: it takes the result from the transformation chain and navigates through the generated solutions. It deals with the common fact that the solvers may produce zero, one or more solutions, and allows the user to create/browse through different solutions until a satisfying one is found. An initial pool of solutions (if any, and possibly just the first) is considered. If the underlying solver chain is incremental, it will be asked to produce additional solutions dynamically when needed.
4. *Standalone launch*: set of plug-ins to enable the standalone execution of the chain, i.e., from command line and without the need to launch Eclipse.

The *SolverChain* (2) contains the core implementation of the approach. The *Launch configuration* (1), *Solution Explorer* (3) and *Standalone launch* (4) components compose the technical backup for the interaction with users/developers.

3.4. Example implementation of the solver-specific part using Alloy/SAT

In order to highlight the main challenges that may arise when implementing a solver chain, we provide an example

[1]MOS GitLab: `https://gitlab.massidia.net/mos/software`

using Alloy/SAT as the solver language/engine. In the following, each step of the process implementation is detailed and inherent issues are discussed. Additionally, the clear separation in different steps allows a modular reuse of our implementation. To this aim, the presented software chain is provided as a set of freely-available independent Eclipse plugins. If a different solver would be used, all the steps in this section should be implemented. [2]

3.4.1. Alloy/SAT

The SAT paradigm has clear limitations: it requires a finite set of boolean variables and only offers a low-level predicate language (only negation, disjunction and conjunction are supported). However, [22] introduced an expressive relational language (Alloy) with a built-in compilation (KodKod engine) that allows the use of many recent SAT solvers. We will thus use Alloy as our target search engine language in order to ease the transformation definition.

Alloy allows for expressing complex predicates using atoms (un-dividable elements), sets (of atoms), relations, quantifiers (universal or existential), operators for relations traversal, etc. However, due to the properties of SAT problems, Alloy cannot be considered as a true first-order logic solver. Indeed, to be able to translate the problem into SAT, a *scope* needs to be given to each typed set, which limits the number of atoms that can be contained in the set.

In Alloy every element is either an atom or a relation, but the language is exclusively based on relations. A relation is a set of tuples, which indicates how atoms are related, with a given *arity*. Indeed, there is no notion of a **set**: a set is represented by an atom, which has a relation that maps to the contents of the set. The main artifacts that we will manipulate in the Alloy language are:

- *Signatures*, declarations of sets, for which the body may contain fields as *relations* to other signatures. Attributes are treated the same as any relation. Scalars, similar to signatures, are treated as sets of atoms. Signatures also support a form of single inheritance.

- *Facts*, declarations of predicates, with quantifiers and an important number of logical, scalar and set operators available.

- *Functions*, which are specific implementations of Alloy built-in functions, such as *max*, *min* or *plus*. The functions may have a direct transformation from the input models, or may need a specific transformation.

[2]We have implemented in [30] translation from the ILOG/CP solver. Despite being based on different concepts, it provided a high-level language abstracting the solver implementation, thus many of the transformation rules had similar structure.

3.4.2. Alloy metamodel and extractor

We developed a metamodel of the Alloy language, which an excerpt is represented in Figure 4 as an ECORE diagram (we have omitted part of the references to improve readability). It is the target metamodel for the generation of the problem definition (task 1) and the partial assignment (task 2). This metamodel shows that an Alloy program is composed by a Module, which is composed by a set of declarations. These declarations may be specialized into: 1) types, where a Signature is a type, composed by Fields, 2) functions, with its corresponding parameters and 3) facts, which are used to express the problem constraints. These facts are written using different kinds of expressions.

We also developed an extractor allowing to produce Alloy textual files from Alloy models (task 3). The generated files help to prototype and to find for errors in the intermediate models. It is implemented using the Acceleo tool[3]. A different option would be to use directly the Alloy Java library, without files generation.

3.4.3. Generation of the problem definition

We divided task 1 into two transformations, respectively from ECORE and OCL, to our Alloy metamodel. They are fully declarative and implemented using ATL [25], which is a framework and language for developing and executing model transformations, transforming source models into target models.

The ECORE to Alloy (*ecore2msalloy.atl*) transformation aims at expressing the structural constraints of a metamodel. An excerpt of the mapping is presented in Table 1.

ECORE concept	Alloy concept
EPackage	Module
EDataType	ExternalType and ExternalModule
EClass	Signature
EAttribute	Field
EReference	Field
EStructuralFeature multiplicity	Multiplicity and/or Fact
EReference containment	Fact
EReference opposite	Fact

Table 1: Excerpt of the mapping from ECORE concepts to Alloy concepts

In this transformation, ECORE classes are mapped to Alloy signatures. Alloy has direct support for abstract and (single only) inheritance. ECORE attributes and references are mapped to Alloy fields. Alloy's fields only support four multiplicity declarations: lone (0-1), one (1-1), some (1-*) and set (0-*). Therefore, other multiplicity lower/upper bounds are turned into corresponding cardinality facts. References properties are turned into facts (i.e., a predicate for the containment or opposite constraint). Finally, attribute's data types are turned into the corresponding Alloy type. Alloy directly supports booleans, integers and strings. Though strings have some

[3]`http://eclipse.org/acceleo`

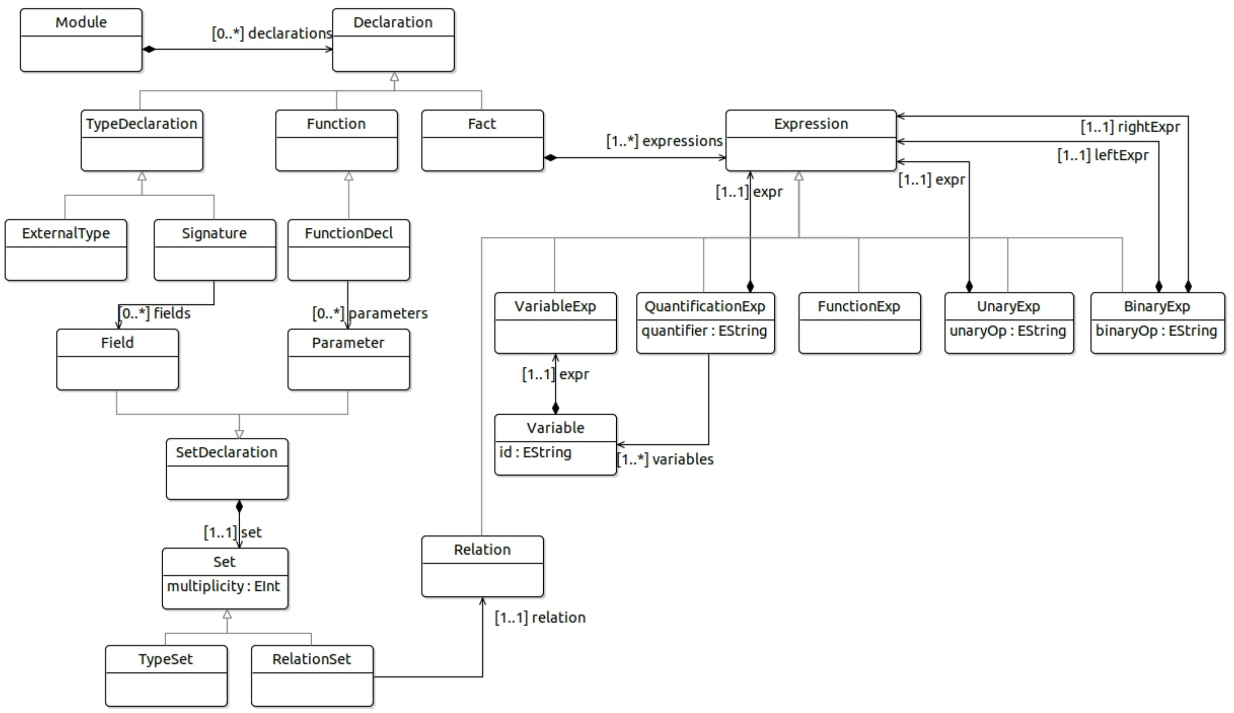


Figure 4: Excerpt of the Alloy metamodel

basic support in Alloy, it comes with several limitations due to the fact that they are treated as scalars: any string usable value must be declared (i.e, Alloy will never generate a string value by itself), and only the equality operation is supported. This solver-specific limitation will be further discussed in Section 3.5.

The OCL/Alloy transformation (*oclmdt-pivot2msalloy.atl*) aims at expressing metamodel invariants as Alloy facts. Concretely, we use the OCL parser offered by the Eclipse project and run the transformation on the resulting OCL Pivot model. An excerpt of the mapping is presented in Table 2, with the corresponding concrete syntax. The output Alloy expression is a composition of concepts; the right column shows only the top-level one. We do not map the entire OCL specification, but only the features that are supported by the Alloy specification. In other words, the input language expressiveness is limited by the solver capabilities. While this may be undesirable, a complete mapping would only be possible if both languages would have equivalent semantics.

The combination of these two (ECORE/Alloy, OCL/Alloy) transformations corresponds to the *CMM2SP* transformation in Figure 2.

3.4.4. Generation of the partial assignment

When the model search chain is ran on a non-empty request M_r, this model has to be turned into a partial assignment for the solver. Here again, we developed a rule-based transformation implemented using ATL. The main difficulty is that the source (search) metamodel MM is unknown to the transformation developer. However, the structural semantics of the model do not depend on this metamodel: they are solely based on the fact that the model contains objects of ECORE classes and may optionally have their structural features (partially) defined. In other words, the type of solver instances and concepts that are created do not depend on the original metamodel. This allows to write metamodel-independent rules in what we called *multi-level* transformation. Its main principles are shown in Figure 5.

The transformation (*model2msalloy.atl*) has three inputs: the (request) model, its (search) metamodel, and the problem definition conforming to the solver metamodel. The latter one is required since the partial assignment is obviously related to its problem definition.

The implementation of this transformation is done using ATL lazy and imperative rules and by accessing the metamodel and model elements using reflection. This is necessary because we do not know the type of the input elements in advance. We are not aware of any technique allowing to implement matched rules (declarative) and that are coupled with reflection, nor of any declarative transformation language supporting this specific matching together with our requirements.

Consider for instance the necessity to transform an input model element into a *Signature* in Alloy. Since we do not know the domain, we develop a transformation rule that transforms an *EObject* into a *Signature*. We list an excerpt of the transformation below with its main aspects.

OCL Pivot main concept [concrete syntax]	Alloy main concept [concrete syntax]
CollectionLiteralExp [Set"a"]	SetExpression ["a"]
IteratorExp (collect) [source->collect(exp)]	NavigationExpression [source.exp]
IteratorExp (forall) [source->forAll(i [: B]| P)]	QuantificationExpression [all i:source | [i in B and] P]
IteratorExp (exists) [source->exists(i [: B]| P)]	QuantificationExpression [some i:source | [i in B and] P]
IteratorExp (one) [source->one(i [: B]| P)]	QuantificationExpression [one i:source | [i in B and] P]
IteratorExp (any) [source->any(i [: B]| P)]	ComprehensionExpression [i:source | P]
IteratorExp (closure) [x->closure(p)]	NavigationExpression [(x.*p)]
OperationCallExp (oclAsType) [x.oclAsType(T)]	OperationExpression [x :> T]
OperationCallExp (oclAsType) [x.oclIsTypeOf(T)]	ComparisonExpression [x in T]
OperationCallExp (includes) [x->includes(y)]	ComparisonExpression [y in x]
OperationCallExp (union) [x->union(y)]	ComparisonExpression [x + y]
OperationCallExp (size) [exp->size()]	SetCardinalityExpression [#exp]
OperationCallExp (min) [A.i -> min()]	IntegerSetFunctionExpression [min[A.i]]
OperationCall (unknown)	ExternalFunction

Table 2: Excerpt of the mapping from OCL to Alloy

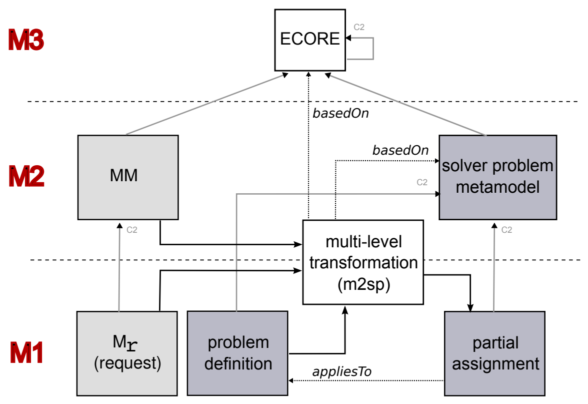


Figure 5: Principles of the multi-level transformation applied to the partial assignment generation

```
unique lazy rule EObject2Sig {
  from
    o: ECORE!EObject
  using {
    attNavExp: MSAlloy!NavigationExpression = OclUndefined;
    attExp: MSAlloy!Expression = OclUndefined;
  }
  to
    oSig: MSAlloy!Signature  (
  id <- o.eClass().name +
           thisModule.classObjectsCounter.get(o.eClass()).toString(),
      -- setting up unique signatures and extending the input class
      multiplicity <- #one,
      "extends" <- MSAlloy!Signature.allInstancesFrom('MMA')->
                        select( s | s.id = o.eClass().name)->first()
do { -- obtaining a given structural feature
  for(att in o.eClass().eAllAttributes) {
    if(not o.refGetValue(att.name).oclIsUndefined()) {
      attNavExp <- thisModule.EStructuralFeature2NavExp(oSig, att);
      if(att.eAttributeType.name = 'EInt')
         attExp <- thisModule.EIntAttribute2EqualsExp(o,att,attNavExp);
      thisModule.CreateFact(attExp);
}
}
-- creates a navigation expression for a pair
-- (object)signature / structural feature
-- i.e. someObjectSignature.someStructuralFeature
rule EStructuralFeature2NavExp(oSig: MSAlloy!Signature,
                               sf: ECORE!EStructuralFeature) {
  to
    navExp : MSAlloy!NavigationExpression (
      leftExp <- objectIDExp,
      rightExp <- sfNameExp
     ),
    objectIDExp: MSAlloy!VariableExpression (
      variable <- oSig.id
    ),
    sfNameExp: MSAlloy!VariableExpression (
      variable <- sf.name
    )
  do {
    navExp;
  }
}


-- creates a comparison expression, to set up the property value
-- i.e. someObjectSignature.someIntegerAttribute =
                        ObjectAttributeIntegerValue
rule EIntAttribute2EqualsExp(o: ECORE!EObject,
         att: ECORE!EAttribute,
         attNavExp: MSAlloy!NavigationExpression) {
to
  equalsExp: MSAlloy!ComparisonExpression (
    comparisonOp <- '=',
    leftExp <- attNavExp,
    rightExp <- attValueExp
  ),
  attValueExp: MSAlloy!IntegerValue (
    value <- o.refGetValue(att.name)
  )
  do {
    equalsExp;
  }
}
```

It contains 3 rules, one for the current element, one creating a navigation expression and one for the specific data type found. The Alloy language does not have dedicated constructs enabling the declaration of a partial assignment, i.e., it does not allow to directly define sets of atoms (to account for existing objects) or relations tuples (to account for existing structural features values). However, this can be circumvented using respectively *unique* signatures and facts.

Singleton signatures are sets that can only contain one atom. For each source object, we thus create a unique signature (*multiplicity* equals to 1) that extends the object's class corresponding signature (shown in the *extends* assignment). A unique name has to be generated for each object's signature, so we use its class name followed by an object counter.

We then create facts (*CreateFact* rule) to account for structural features and its values (this rule is not presented at the transformation excerpt, since its implementation is simple). It receives a navigation expression that depends on the input object type. But, for a given object, we do not know the names of its structural features, so it is necessary to create a loop over all the attributes and to obtain their values through reflection (*refGetValue* method). We create an equality navigation expression: the left side has the object variable and its attribute name (*EStructuralFeature2NavExp* rule); the right side has the given attribute value, which can be any data type. In this example code, we show the conditional expression for an Integer value (*EInt* type), where we call a specific lazy rule (*EIntAttribute2EqualsExp*). Each kind of attribute need to have one lazy rule (e.g., for *EString*, *EBoolean* or other objects). The same kind of loop is implemented for the object references.

Additionally, the transformation accepts a model search option that allows to *freeze* structural features of objects. This is realized by creating an additional constraint on the field for the object's corresponding unique signature, which cardinality must then be equal to the number of actual values in the original structural feature.

Similarly to task 1, the resulting partial assignment is processed by our Alloy extractor which generates its corresponding Alloy file.

3.4.5. Solver execution

Now that the problem definition and optionally a partial assignment have been generated, we may ask the solver engine to calculate solutions (if any). For this, we generate a master Alloy file containing the Alloy *command* to be executed, and the necessary imports to all previously generated files. The solver execution in Alloy needs 4 model search options: *scope*: the amount of used elements; *SAT solver*: the SAT solver to be used, e.g., *SAT4J*; *Bitwidth*: the integers allowed range and *generate a single module*: to generate only one Alloy module with all the specifications. If any scope was given, its class bounds are added to the command parameters as specified by the Alloy language. Otherwise, Alloy will use its default global bound. Depending on the used solver, its solving capacities may differ according to the implementation. For instance, a specific solver could provide support to (non)monotonic operations.

3.4.6. Generation of the solution model

A model search user expects the output(s) as model(s) conforming to the original search metamodel MM. The generation of these solution model(s) corresponds to task 5 of our example process.

The Alloy/SAT solver may generate more than one solution, depending on the specified models and constraints. When this happens, we take the first generated solution to generate the solution model. However, the user may iterate over any of the generated solutions, and generate the corresponding solution models. Each time the solver generates a new solution, task 5 is divided into two subtasks (as shown in Figure 2). First we use EMF's built-in features to generate a model conforming to Alloy's XML schema metamodel.

The second subtask ($SS2M$) in Figure 2 is the transformation of the resulting model into a solution model conforming to MM. Again, we implemented it using ATL (*MSAlloyInstance2Model.atl*). Similarly to the partial assignment generation task, the difficulty here is that the target metamodel is unknown to the transformation developer. We therefore apply again the multi-level transformation technique used for task 2. Its application to the generation of a solution model, independently from a specific solver, is illustrated in Figure 6. It receives as parameters the solver solution model and also the metamodel of the output solution, in order to discover the type of the elements that need to be generated.

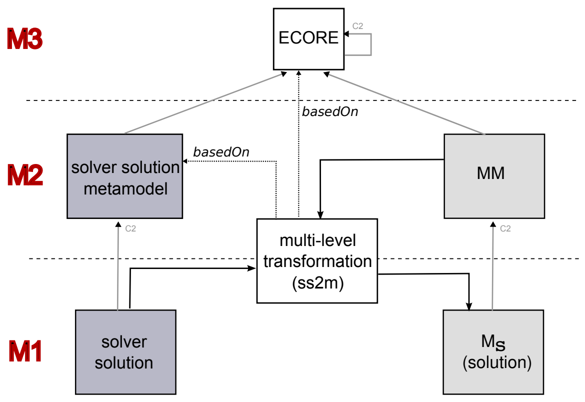


Figure 6: Principles of the multi-level transformation applied to the solution model generation

We need to consider two types of elements from the input model: atoms and relations tuples. Atoms, when they belong to the set of a signature that was generated from a class, yield class objects. Other atoms can be either datatype values (which use a specific signature) or built-in elements from the Alloy specification.

Relation tuples yield the assignment of an object's structural feature value, which name is the relation name. The first element of the tuple is always an atom that corresponds to the source object. The second element can be either a class object atom (in the case of a reference target), or a datatype value atom (in the case of an attribute value).

As a side implementation note, it can be noted that the first rule creates an object whose type can only be known during rule execution. The second rule assigns a structural feature value to an object created by another rule. ATL matched rules do not support creating an object with a dynamically computed class, nor does it support assigning properties to objects created in another rule. Therefore we again use some ATL imperative constructs in the transformation. To the best of our knowledge, there is no existing declarative transformation language supporting these features.

3.5. Example and results

To illustrate the model search process, we provide results on two use cases: a class diagram (CD) generation example [34] and a hierarchical state machines (HSM) example [35], where the specifications were adapted from literature to be implemented in our framework. In both cases, the goal is to set up a simple metamodel and then to generate a solution with specific properties given by a set of constraints written in OCL. There is only one possible solution for a given number of classes/states. Our objective here is not to study absolute performance but rather the general behaviour: indeed our example Alloy/SAT solver chain only illustrates the approach and does not include any special solving optimization.

We show the class diagram metamodel in Figure 7. It states that the generated solution will have only packages, classes and attributes.

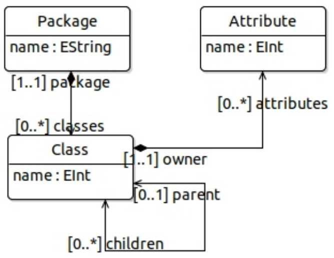


Figure 7: Class Diagram metamodel for Model Search

The additional OCL constraints for the instances generation are shown below. The goal is to generate attributes with different names withing the same class, by numerically increasing the name of the attributes. The complete Alloy program generated for this specification is listed in Appendix A.

```
context Class inv differentNames:
  Class.allInstances()->excluding(self)->
    select(c | c.name = self.name)->size() = 0
context Class inv hasOneAttribute:
  attributes->size() = 1
context Class inv noParentRelationship:
  parent->size() = 0
context Attribute inv attributeName:
  name = owner.name
context Attribute inv attributeNamePositive:
  name >= 0
context Class inv classNamePositive:
  name >= 0
```

We show in Figure 8 the metamodel used for the HSM example. The metamodel is formed by a root state machine, which can be formed by simple or composed states. They are connected by incoming or outgoing transitions.

Their corresponding constraints are illustrated in the following. The constraints are more complex than in the class diagram example, since they need to guarantee the nested structure, the state machine and transition names, and that transitions need to connect to one ongoing and one outgoing state. The explanation of each constraint is done in the code comments.

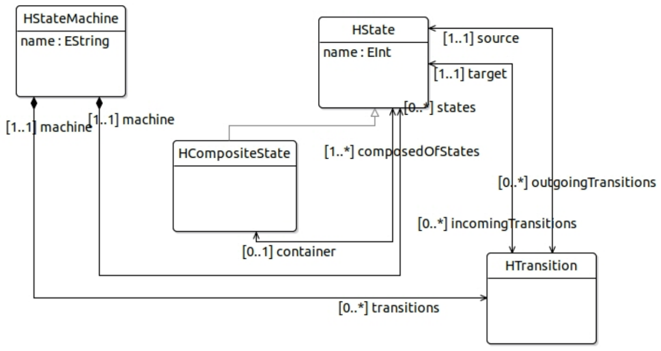


Figure 8: HSM metamodel for Model Search

```
-- prevent states self inheritance
context HState inv inheritance:
  not self->closure(container)->excluding(self)->includes(self)

-- positive names
context HState inv positiveName:
  name >= 0
-- names limited to overall number of states
context HState inv maxName:
  name <= HState.allInstances()->size()
-- different names for all top states
context HState inv differentNames:
  container->size() = 0 implies HState.allInstances()->excluding
  (self)->select(s | s.container->size() = 0)->select
  (s | s.name = self.name)->size() = 0
-- limit top-states name to number of top-states
context HState inv maxTopName:
  container->size() = 0 implies name <= HState.allInstances()->
  select(s | s.container->size() = 0)->size()
-- names of top states span over all integers
context HState inv incNames:
  self.name > 0 implies HState.allInstances()->select
  (s | s.container->size() = 0)->select(s | s.name = self.name-1)
  ->size() > 0
-- name = container -1 for non-top states
context HState inv nonTopNames:
  self.container->size() = 1 implies name = container.name - 1
-- top-states (except first/last) are composite
context HState inv topIsComposite:
  container->size() = 0 and self.name > 0 and HState.
  allInstances()-> select(s | s.name> self.name)->size() > 0
  implies HCompositeState.allInstances()->includes(self)
-- limit hierarchy size to state name
context HCompositeState inv limitHierarchySize:
  self->closure(composedOfStates)->size()-1 <= name
-- all but last top-state must have full hierarchy
context HCompositeState inv setHierarchySize:
  container->size() = 0 and HState.allInstances()->select(s |
  s.name > self.name)->size() > 0 implies self->
  closure(composedOfStates)->size()-1 = name
-- overall number of transitions
context HStateMachine inv nbTransitions:
  HTransition.allInstances()->size() = HState.allInstances()->
  size() - 2
-- prevent self transitions
context HTransition inv noSelfTransition:
  not (self.source = self.target)
-- each state has at most one outgoing transition
context HState inv atMostOneOutgoingTransition:
  outgoingTransitions->size() <= 1
-- transitions have source and target with same name
context HTransition inv equalST:
  self.source.name = self.target.name - 1
-- transitions have source and target in different hierarchy
context HTransition inv differentHierarchy:
  not self.source->closure(container)->includes(self.target)
```

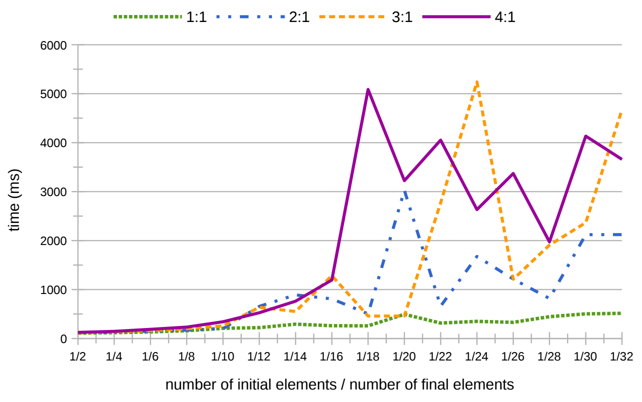
These two examples are injected into the model search chain to be executed by the solver. Note that for each example, it is necessary to inject the metamodel and its instances (partial assignment). The generic multi-level transformation avoids having one transformation *CD2Alloy* and another one *HSM2Alloy*.

As in the original example, we observe the computational behaviuor when gradually increasing the number of elements (i.e., instances of the metamodel classes) in the requested solution, from 2 to 32 (CD) or 26 (HSM) elements. Here we always start from the same initial model containing only a root element. We first provide experiments on the class diagram generation problem with different global scopes (a mandatory parameter for Alloy and most existing solvers) to observe the impact on performance. Then we compare results on the same problem using either the SAT4J or MiniSAT back-end.

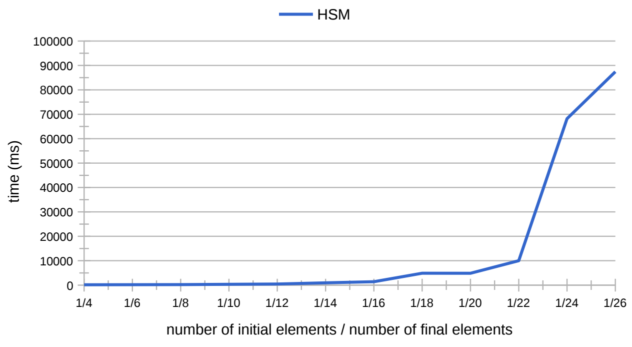
In all these experiments we focus on the model finding times, excluding the problem (resp. solution) generation (resp. extraction). Indeed, due to exponential combinatorial search, the former is a decisive factor in the overall computational behaviour. The latter, pseudo-linear in practice, soon becomes negligible (plus the problem only needs to be generated once). We provide the average values based on 20 runs of each problem on a 16Gb Xeon 3.3Ghz linux computer using Alloy 4.2. Results are summarized in Figure 9. The examples and full experiments are available for reproduction in [2].

The top figure presents results with different scopes using MiniSAT with a fixed integer bitwidth (5). The ratio 2:1 denotes that the global scope is set to twice the number of requested classes. These results show that the scope has an important impact on the computation times. Indeed a higher scope results in a higher number of boolean variables in the generated SAT problem thus potentially inducing a higher number of branches to explore (despite mitigation by symmetry breaking). Although a smaller scope is in general better for performance, our experience shows that choosing the right scope is tedious and largely problem-dependant. Similar effects can be observed on problems with numerical constraints when varying the integer bitwidth (which defines the range of integer values). Since Alloy translates integers to sets of boolean (one for each value), this again results in a higher number of variables and thus a similar combinatorial impact. Finally, in most of our experiments, we can observe occasional gaps (resp. peaks) caused by specific instances being easier (resp. harder) for the solvers built-in optimizations.
The bottom figure shows the results for our state machines generation example. We defined complex constraints that mix integer and set computations so that the hierarchy size for top states increases with the problem size. We can indeed observe a higher computational cost than for the class diagrams generation example.

These experiments indicate that Alloy/SAT may not be the best solver choice for a number of problems, particularly those involving various numerical constraints or those where it is hard to guess the approximate number of instances in the solution. This confirms the interest of having a solver-agnostic model search approach,



(a) Generation of class diagrams: varying scope ratio (bitwidth=5, MiniSAT)



(b) Generation of state machines (fixed scope, MiniSAT)

Figure 9: Results on the generation problems

allowing users to select a method depending on their problem characteristics. We believe that, generally, model search would benefit from a solver that does not impose scope restrictions. Unfortunately, efficient solvers which allow on-the-fly instance creation are not convincingly demonstrated in the literature. Finally, although this is more obvious on complex problems or on large scales, an exponential behaviour can always be observed when the size of the problem grows. This is expected from a combinatorial solver but limits usability when using a SAT-based solver that can only cope with small to medium size problems [22].

4. Transformation as search

In the following we present our generalization of model search to model transformation/update operations by considering multiple metamodels (sources and/or targets) together with the transformation specification as a single model search input. This operation is called *transformation as search* (TAS). The main idea is to define the transformation/update as a set of relations and constraints between elements of the metamodels that are to be related (these may be called *weavings*). All these artifacts are then unified into a *transformation metamodel*. By applying model search on this unified metamodel, a model which *contains* target model(s) is created. A major feature of the approach is that the operations which yield the model search problem are completely solver-independent, thus allowing to directly use any underlying model search implementation. Additionally, since TAS solely uses basic modeling elements (metamodels and constraints), no specific transformation language is introduced. However, one could define a higher-level language to ease the writing of specifications (or use an existing language) and them implement a translation into a weaving metamodel and its constraints.

Another important feature is that the operation is inherently bi-directional and incremental: the same specification can be used to generate any (or extend previously existing) weaved models. Indeed, in a TAS operation, source/target metamodels are treated equally in the specification and only make sense once a given transformation scenario has been requested. In the following, we will thus call the weaved metamodels *input* metamodels instead of source/target metamodels. Finally TAS is not restricted to one-and-one scenarios: any number of input metamodels can be weaved within a single specification.

In the next subsections, we first introduce a running example and three possible scenarios. We then formally define the TAS operation and the different steps involved, with illustrations on the first classical scenario (creation of a target model from a source model). We then present how TAS can be applied to the running example for the two other scenarios: reverse transformations and updates. Finally, we describe our TAS implementation and provide some experimental results.

4.1. Running example

The chosen example is a transformation between a class schema model (MM^{CS}) and a relational schema model (MM^{RS}), known as the *Class2Relational* transformation. We have chosen this use case as illustration because it is well-known and rather simple (allowing the reader to quickly grasp the involved domain concepts) and has been studied in other works to demonstrate different aspects about transformation languages (such as [40], [33], and others). The transformation input metamodels are presented at both sides of Figure 10 (some elements have been omitted to improve readability).

The first scenario is the traditional creation of a relational schema (the target model) from a class schema (the source model). The second scenario is the reverse transformation: creation of a class schema from a relational schema. The third scenario is an update: both models pre-exist, then the class schema is modified and the relational schema needs to be updated accordingly.

We will apply the scenarios on a "Family" class schema illustrated at the top of Figure 11. The bottom part is a relational schema created by the first scenario, and is also the source for the second scenario.

4.2. Transformation as search process

The complete TAS process is illustrated in Figure 12. It consists of three main steps: creating the model search problem, running the search, then isolating the target models from the solution model. Creating the model search is itself composed of two subtasks: creating the search metamodel (i.e., the problem definition, here called the transformation metamodel) and creating the search request (i.e., the partial assignment, here called the transformation request). Each of these tasks is formally defined and detailed in the following along with its illustration on our example's first scenario.

4.2.1. Obtaining the transformation metamodel by unification

The first step is to obtain a transformation metamodel, called CMM^T, by *unification* of the input ($\{CMM^0, \ldots, CMM^n\}$) and weaving (CMM^W) metamodels. This part of the process is independent from the chosen transformation scenario.

In our example, these are respectively the class schema structure (left part of Figure 10), the relational schema structure (right part), and a set of weaving elements and constraints (middle part, constraints are not shown in the Figure). The application of this operation to our example is illustrated in Figure 13. Its result is the whole Figure 10.

Metamodel unification is a simple operation, consisting merely in copying and combining the inputs into a new metamodel. Formal definitions of CMM^W and CMM^T are given below:

Definition 11 (weaving metamodel). *We call* weaving metamodel *between* $\{CMM^0, \ldots, CMM^n\}$*, a constrained metamodel* CMM^W *defined by* $CMM^W =< MM^W, C^W >$*, where* MM^W *and* C^W *are respectively a set of metamodel elements and constraints that define the weaving relationships between the elements of* $\{CMM^0, \ldots, CMM^n\}$.

In ECORE, the weaving metamodel targets the input metamodels elements through the use of cross-model references.

Definition 12 (transformation metamodel). *Let* CMM^W *be a weaving metamodel and* $\{CMM^0, \ldots, CMM^n\}$ *the set of weaved metamodels. We call* transformation metamodel *the constrained metamodel* CMM^T *defined by* $CMM^T =< MM^T, C^T >$*, where* $MM^T = MM^0 \cup \ldots \cup MM^n \cup MM^W$ *and* $C^T = C^i \cup \ldots \cup C^n \cup C^W$*. The operation consisting in obtaining* CMM^T *is called* metamodel unification.

Obviously, in ECORE, metamodel unification turns cross-model references in the weaving metamodel into intra-model references in the transformation metamodel.

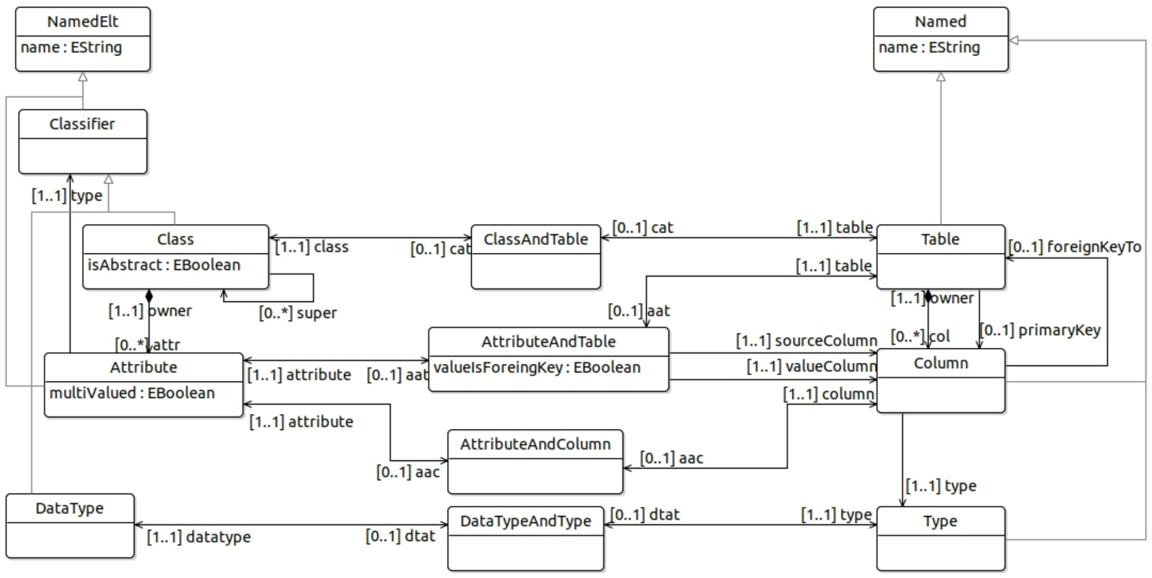


Figure 10: Extract of the running example transformation metamodel as an ECORE diagram. Input metamodels are on the sides, weaving metamodel is in the middle.

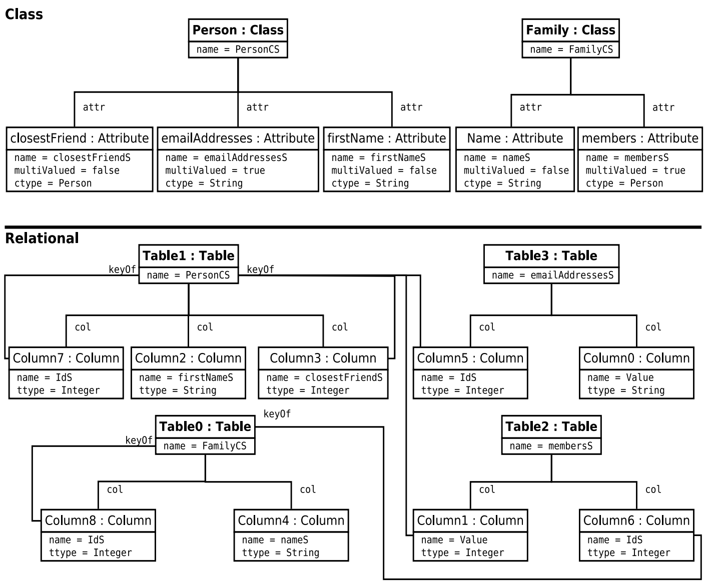


Figure 11: Source and target models from the running example (scenario 1) as instance diagrams

4.2.2. Creating the transformation request by unification

The next step is to define the transformation request (which will act as the model search request). The request depends on the chosen scenario, which is defined by setting a *behaviour* on each model of the input metamodels. Three behaviours are supported: *generate*, *freeze* and *extend*. These behaviours allow the selection of a scenario by specifying which models are part of the request and which should be generated. Also whether or not they can be modified in the final solution. "generate" means that the model does not yet exist and should be created by the transformation. In other words, it is a target model created from scratch. "freeze" means that the model exists and should not be modified by the transformation. In other words, it is an immutable source model. Finally "extend" means the model exists but may be modified by the transformation. In other words, it is both a source and a target model. The different combinations of these behaviours give birth to the potential scenarios. In our example's first scenario, the classical Class2Relational transformation, the class schema is set to "freeze" while the relational schema is set to "generate".

The transformation request is obtained by unification of all source models and optionally a weaving model (the latter is a previous transformation trace and can be used for update scenarios).

Definition 13 (transformation request). *Let CMM^T be a transformation metamodel created from a set of input (and weaving) metamodels $S = \{CMM^0, \dots, CMM^n, CMM^W\}$. Let $s = \{M^0, \dots, M^p, M^W\}$ be a set of source (and weaving) models (where $\forall M^i \in s$, M^i conforms to $CMM^i \in S$). We call* transformation request *for CMM^T the model M_r^T defined by $M_r^T = M^0 \cup \dots \cup M^p \cup M^W$. The operation consisting in obtaining M^T is called* model unification.

The definition above encompasses all scenarios to depict the unification. However, some of the models can be empty/absent, depending on the scenario. This means the weaving model is optional when defining one new instance, but it is always used or generated if not existing. In our example's first scenario, there is only one source model and no previous trace, the transformation request therefore simply consists in a copy of the class schema model elements, which is the "Family" class schema at the top of Figure 11.

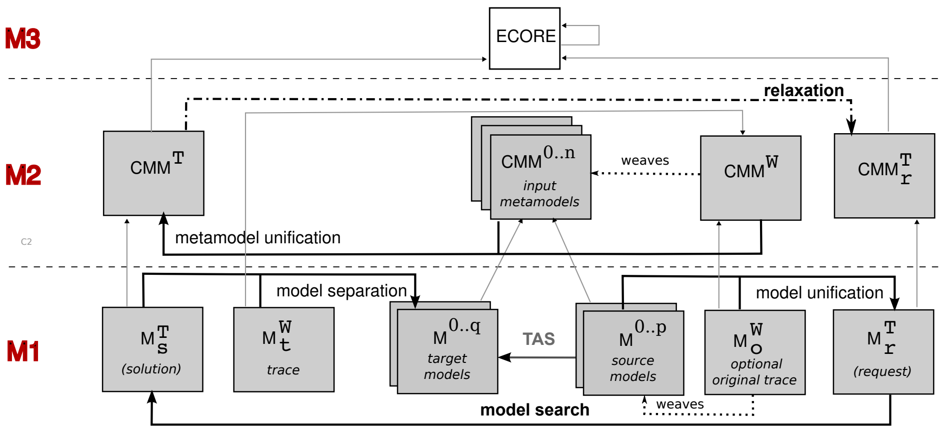


Figure 12: Transformation as search process

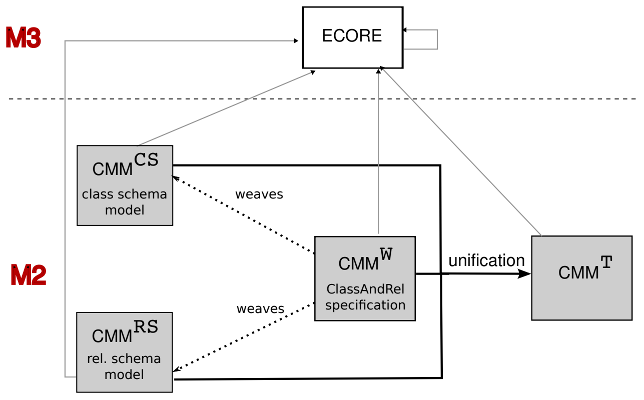


Figure 13: Obtaining the example transformation metamodel by unification

4.2.3. Running model search

The next step is to run model search on the previously defined problem. From definition 10, a valid model search request (here the transformation request) must be a *partial model* of the search metamodel (here the transformation metamodel CMM^T). For any TAS problem, this property is ensured by the following proposition:

Proposition 1. *Let CMM^T be a transformation metamodel. Any transformation request for CMM^T is a* partial model *of (or p-$conformsTo$) CMM^T.*

Proof 1. *From definition 9 of p-$conformsTo$, it resolves to finding a relaxed metamodel $CMM_r^T =< MM_r^T, C_r^T >\in Rx(CMM^T)$ such that M_r^T $conformsTo$ CMM_r^T. From definition 7 of conformance, this requires that (1) M_r^T $conformsTo$ MM_r^T and (2) $M_r^T \in C(MM_r^T)$.*
Let CMM_r^T be the relaxed metamodel of CMM^T such that $MM_r^T = MM^T$ and $C_r^T = \emptyset$ (i.e., the one obtained by removing all constraints). (2) is obviously true since there are no predicates to satisfy. (1) requires that MM_r^T can be a reference model of M_r^T, i.e., its graph G_r^T contains all nodes (meta-elements) targeted by the graph g_r^T of M_r^T. Let $S = \{MM^0, \ldots, MM^p, MM^W\}$ be the input and weaving metamodels. (1) is clearly true since on one hand, by definition 12 of CMM^T we have $\forall MM^i \in S, MM^i \subset MM^T$ (in particular $G^i \in G^T$), and on the other hand $MM_r^T = MM^T$ (in particular $G_r^T = G^T$).

In other words, since each input metamodel (as well as the weaving metamodel) is a subset of the transformation metamodel and each source model conforms to an input metamodel, any transformation request p-$conformsTo$ to the latter.

The model search operation extends the transformation request M_r^T into a solution model M_s^T that conforms to CMM^T (when there are solutions). By satisfying weaving constraints, search thus produces a solution model which contains both source/target model elements and weaving model elements (these can be understood as the transformation traces). Additionally, model search ensures that models satisfy their own metamodel constraints, effectively preventing the creation of ill-formed target models. In our example's first scenario, the solution model, without the transformation trace, is shown in Figure 11.

4.2.4. Obtaining the target models and transformation trace by separation

The final step is to isolate the target models contained in the solution model as independent models. This operation, the reverse of model unification, is similarly a simple operation: for each target metamodel MM^j, it suffices to copy all elements from M_s^T that are associated to MM^j into a new model. The same technique can be applied to MM^W in order to obtain the transformation trace as an independent model. For the latter, in ECORE, weaving's intra-model references are therefore turned into cross-model references targeting the previously separated target models or the original source models.

In our example's first scenario, a sample target model result is the "Family" relational schema composed only of the elements illustrated at the bottom of Figure 11.

4.2.5. Application on other scenarios

As previously mentioned, different scenarios are obtained by varying the transformation request through the possible combinations of models behaviours.

For our example's reverse transformation scenario, behaviours are exchanged, i.e., the existing relational schema is set to "freeze" (source model) and the class schema is set to "generate" (target model). By applying the same process, a class schema will be created. Note that depending on the transformation specifications, the operation is not necessarily bijective since it is not even injective in the general case. However the original class schema is necessarily among the potential solutions.

For our example's update scenario, the class schema is set to "freeze" while the relational schema is set to "extend". As a consequence, the relational schema will be updated to maintain consistency with the class schema based on the weaving constraints. If the original schemas were obtained by a transformation, its trace can be provided, effectively forcing the transformation to maintain previous mappings. If a trace is not provided, TAS will recreate mappings for all elements. These may be different ones if the specifications allow it, though again the original mapping (if any existed) is necessarily among the solutions.

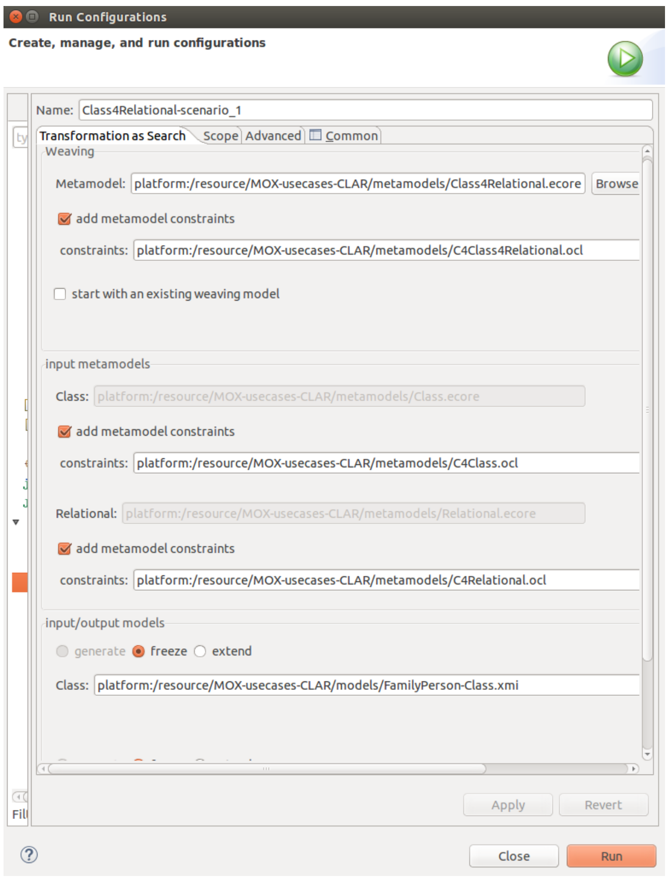
Other scenarios are possible. By setting both models to "freeze", TAS checks whether it is possible to map two given schemas and the potential mapping is provided in the transformation trace. By setting two existing schemas to "extend", it will allow to recover consistency by modifying any (or both) model(s).

Finally, we do not present an example with more than two input metamodels (which means more than one source or target model) since it does not introduce any difference for the TAS process: any number of input metamodels can be weaved by a specification while applying the exact same process. In other words, TAS is not limited to one-and-one transformations/updates.

4.3. Implementation

The transformation/synchronization (UI and API) software parts implement the complete TAS chain illustrated in Figure 12. Again, it is freely distributed as a set of Eclipse plugins under EPL license [2].

The input, illustrated on the running example as a UI screen-shot in Figure 14, is a TAS specification: the weaving constrained metamodel, the involved constrained metamodels, the optional input and trace models, and finally the choice of model behaviours (which define the scenario being requested).


Run Configurations
Create, manage, and run configurations

Name: Class4Relational-scenario_1

Transformation as Search | Scope | Advanced | Common

Weaving

Metamodel: platform:/resource/MOX-usecases-CLAR/metamodels/Class4Relational.ecore

[✓] add metamodel constraints

constraints: platform:/resource/MOX-usecases-CLAR/metamodels/C4Class4Relational.ocl

[ ] start with an existing weaving model

input metamodels

Class: platform:/resource/MOX-usecases-CLAR/metamodels/Class.ecore

[✓] add metamodel constraints

constraints: platform:/resource/MOX-usecases-CLAR/metamodels/C4Class.ocl

Relational: platform:/resource/MOX-usecases-CLAR/metamodels/Relational.ecore

[✓] add metamodel constraints

constraints: platform:/resource/MOX-usecases-CLAR/metamodels/C4Relational.ocl

input/output models

generate [✓ freeze] extend

Class: platform:/resource/MOX-usecases-CLAR/models/FamilyPerson-Class.xmi

Figure 14: TAS launch configuration screenshot

The TAS designer, i.e., the user of the framework that will execute a TAS operation, has to create a set of minimal artifacts for performing model transformations, i.e., a TAS specification. It needs to specify the source and target metamodels and the weaving metamodel, to be able to create relationship between the elements. The metamodels need to be written in ECORE. In addition, it is also necessary to create at least one source or target model, depending on the direction of the transformation. The 3rd scenario (synchronization) is also interesting for illustrating the *multi-level transformation*. Consider the need to translate the instances of the *Class* and *Relational* metamodels into the solver format: it is not necessary to develop *Class2Alloy* or *Relational2Alloy* transformations.

Other artifacts may be created as well, to obtain a more precise specification, such as the OCL constraints over the source and target metamodels. Finally, an already existing weaving model may also be set up as parameter, as in the synchronization scenario. It will enforce the existence of the already created relationships.

We have opted not to create a new transformation language or to use existing ones, such as ATL or QVT. The transformations are created only using models/metamodels and OCL constraints. However, these languages could be integrated into the framework, by developing the transformations for the ATL or QVT specifications into our Model Search layer. To implement this, the transformation language would need to have a transformation model and a metamodel.

The TAS specification is then translated into a *Model Search* problem following the metamodel and request model unification steps previously presented: the search metamodel (which yields the problem definition) is set to the generated transformation metamodel, and the search root model (which yields the partial assignment) is set to the generated transformation request. These steps are implemented using EMF's Java API. Additionally, when a model behaviour is set to "freeze", two elements are added to the resulting model search specification. First, the model search scope sets each class bound to the number of corresponding instances in the frozen source models. However, this is not sufficient since structural features may still be modified. A second option is set, presented in subsection 3.4.4, which freezes attributes and references of the corresponding frozen parts of the root model.

In order to obtain the output(s), our TAS implementation provides a solution explorer interface that embeds the model search solutions explorer. Each time a new solution is requested, the underlying model search problem is solved using the chosen solver chain. As previously presented, the solution (if any) is separated to obtain the target models and the transformation trace. Again, these solver-independent steps are realized using EMF's Java API, similarly to (meta)model unification. The current implementation saves all the intermediate files generated during the process, such as the solver specifications, the solution model, the transformation model and metamodels, and others. This enables a detailed analysis of each execution step of the TAS chain.

4.4. Results

We provide results on the running example and an additional specification taken from the literature. Experimental conditions are the same as to the ones presented in Subsection 3.5. Similarly, we do not aim at evaluating absolute performance (which highly depends on the chosen solver chain) but rather focus on the differences between scenarii, specifications and problem instances. Therefore we provide results with a fixed backend (Alloy/MiniSAT), scope ratio (3:1), and bitwidth (5). All these experiments are available for reproduction in [2].

4.4.1. Results on the running example

Scenarii, specifications and problem instances. We executed 3 distinct scenarii:

- The forward transformation presented in the running example (from class to relational). In this case, we set to *freeze* the left model and to *generate* the right model, which will be generated from scratch.
- the reverse transformation (from relational to class). We configure to *freeze* the right model produced from scenario 1, and we produce a new left model.
- a synchronization scenario (propagation of changes from one model to another after adding one new element to the class metamodel). We set *freeze* to the left model, *extend* to the right model, and provide the trace from the first scenario as the starting weaving model.

The specifications, that is the three metamodels and their constraints, are exactly the same for the three scenarii. In order to illustrate how transformations are specified in our approach, we provide below some example constraints that apply on the transformation together with a brief explanation. These constraints, together with the input metamodels and models are translated into an Alloy program. The complete list of generated constraints, in Alloy, are listed in Appendix B.

```
-- c.1
context Attribute inv:
 owner.attr->forAll( a : Attribute |
  not(self = a) implies not(name = a.name))
-- c.2
context Class inv ClassTable:
  cat->size() = 1
-- r.1
context Table inv:
  primaryKey->size() = 1
  implies primaryKey.type.name = 'Integer'
-- c4r.1
context ClassAndTable inv:
   class.name = table.name
-- c4r.2
context AttributeAndColumn inv:
   attribute.owner.cat.table.col
    ->includes(column)
-- c4r.3
context AttributeAndTable inv:
   sourceColumn.foreignKeyTo
    = attribute.owner.cat.table
```

- *c.1*: prevents a class from having several attributes with the same name. Applies to any (source or target) Class model;
- *c.2*: specifies the cardinality of the link, i.e., that a Class is linked with only one Table;
- *r.1*: sets the type of a primary key column to "Integer". Applies to any (source or target) relational model;
- *c4r.1*: weaved classes and tables must have the same name;
- *c4r.2*: for single-valued attributes, weaved attributes and columns must have the same owner, i.e., their containing *Class* and *Table*;

- *c4r.3*: the source column of an N-N relationship must be a foreign key to the attribute's owning class.

Writing OCL constraints for TAS specifications is very different from writing classical rule-based transformations. Indeed, the goal here is to narrow the set of possible solutions to the acceptable ones by incrementally adding constraints. If one so wishes, it is possible to use a dedicated language (such as QVT-R) that would then be translated to a weaving constrained metamodel. Our approach is very expressive, since the approach is not dependent of the top transformation language, but on the definition of the weaving models. The choice of a given transformation language, or a subset of it, is a trade-off between expressiveness and usability. The iterative process of creating weaving specifications often shows the modeler that source/target metamodel constraints had not been fully specified, for instance by proving that badly-formed models can be created. However, discussing the details of this implementation is out of scope of this paper.

Finally, we provide experiments on two different problems while gradually increasing the instances size: the instances we previously generated using model search; and the family-person custom diagram depicted in Figure 11 for which we gradually add attributes to one class.

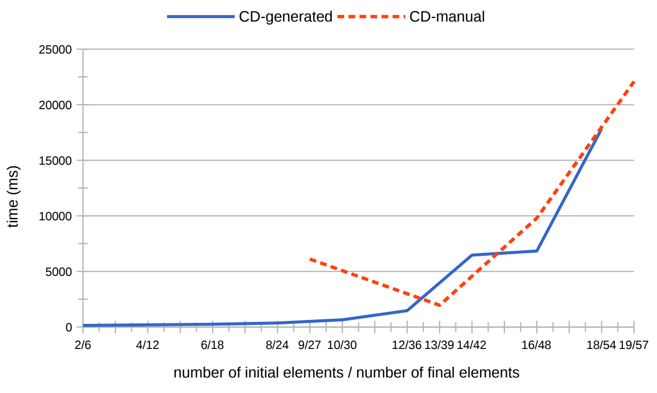
Results. Figure 15 summarizes the results for this set of experiments.

First, we can obviously note the difference of magnitude between the different scenarios. For the generated diagrams, scenario 1 tops at 18s, scenario 2 at 113s and scenario 3 at only 6s. The latter is easy to explain. Indeed, while the global number of final elements is the same, most of them already exist and are weaved in the original trace (this can be seen by looking at the initial number of elements, i.e, the given partial assignment size). Thus only the added elements create search decisions resulting in a slower computational increase along with the problem size. This may indicate that the approach is particularly fitted for the update scenario. The differences between the first two scenarios are the result of the specifications producing a higher number of possibilities to explore on the reverse scenario. On these two scenarios the exponential behaviour expected from combinatorial search is observed. This again confirms[22] that the Alloy/SAT solver, at least without any special optimizations, is only efficient on small to medium problem sizes.

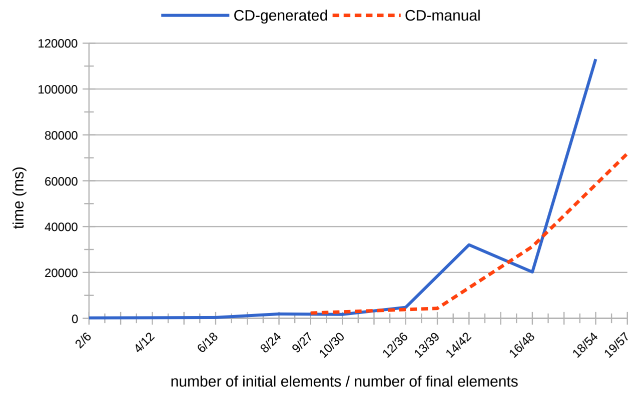
We can also observe that in every scenario the computation times are similar between the generated and manual instances, though the latter only grows in the number of attributes. This may indicate that difficulty depends mostly on instance sizes and not on the particular shape of instances.

4.4.2. Results on the literature state machines example

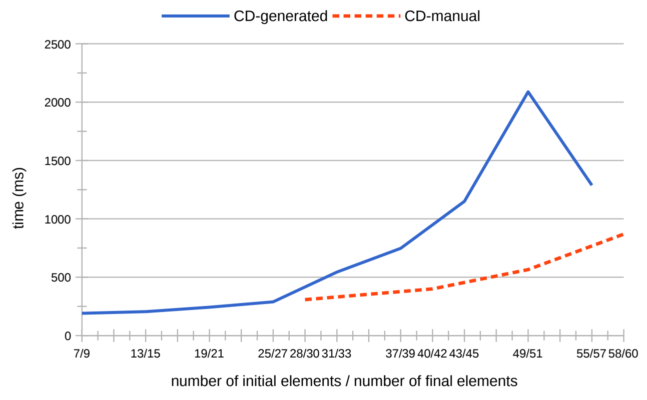
We adapt the example proposed by [35]. The example involves transforming hierarchical state machines to non-hierarchical ones using the OCL *closure* operator in the constraints. Figure 16 summarizes the results on the first scenario so as to compare with the running example. As can be seen on the Y-axis, the computational cost is much lower than for the generation of relational databases from class diagrams, topping at only 0.34s for the same number of target elements. This confirms that difficulty is largely dependent on the problem specifications. Indeed this scenario only requires the solver to unfold the states hierarchy while preserving existing transitions.



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

Figure 15: Results on the running example

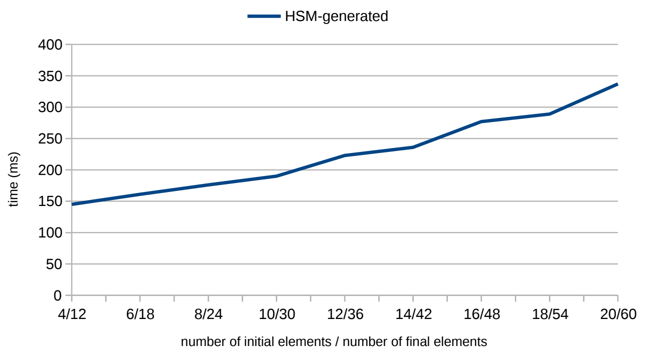


Figure 16: Results on the state machines example

4.4.3. Results summary

The approach is functional but suffers from computational issues. Obviously, computation times are highly impacted by the specifications. The running example strongly constraints the result, while the state machines example is clearly an easier problem. The instances size and the chosen scenario are also an important factor. This combinatorial approach seems generally more fitted for synchronization than classical transformations or generations. Finally a number of limitations stem from the chosen solver backend. Indeed, the Alloy/SAT combination shows its limits both in the size of problems it can handle, which can be mostly tied to the required scope issues, and in the types of operations that can be used in the specifications. Indeed arithmetic operations are supported but costly, while string manipulations are strictly limited to plain equality. We believe this confirms the relevance of our solver-agnostic abstraction. On one hand it eases the development of alternative backends since only a part of the model search chain needs be defined. On the other hand it allows the user to switch backends based on specific needs or even break down problems into different sub-parts and solvers.

5. Related work

We describe the related work classified into three groups: first, the approaches focusing on model finding and model generation; second, the ones focusing on coupling transformations and optimization; finally, the solutions covering model relations and transformations as search, which are the closest to our approach. It is important to note that the comparison and description of approaches focuses on its overall components and chain, the supporting framework and the interactions between them, not on the specific details on the translation from the input language (e.g., ECORE/OCL) to each solvers' (e.g., Alloy/SAT) format.

Model finding/generation

Several works have studied the benefits of model finding/generation within a modelling environment. Among the first ones is USE [17], which applies a custom engine on UML/OCL for validation and later use case generation purposes. A large number of approaches, including ours, follow the same principle of translating a given modeling specification into the solver format, but applied in other context and with distinct capabilities. Model repair approaches (see a survey in [38]), such as [19] use a similar technique to decrease the number of inconsistencies in a given model. A number of other solver-specific approaches have followed: [6] transforms UML/OCL to a CSP solver specification for validation, others target the Alloy/SAT language/solver [4, 39, 37, 9].

In our approach, we explicitly describe the needed chain of transformations implemented by the framework, covering the injection/solving/extraction operations. We also provide explicit definitions of partial, relaxed and constrained metamodels/models, which are present on related approaches but not always defined. The definitions are adapted to represent enumerative solutions, where it is necessary to have a pool of elements already existing. The generative ones are also supported, following a more typical modeling scenario. For instance, these definitions handle the existence of ECORE models elements that do not need to be connected and constrained by all the input metamodel definition. The translation of ECORE/OCL to Alloy from our solution is similar with previous work. As drawback, we do not provide deeper studies on solver specific characteristics that could improve performance and quality of the solutions.

Transformations and optimization

A couple of solutions from this category have the final goal similar to ours, which is to perform model transformations and model generation. However, the approaches main contributions focus on using and/or improving optimization techniques in a MDE technical space, not on providing a framework. [37] offers two notable features: a measure of similarity to compare search solutions and an automated reduction of the resulting Alloy Formulas to decrease computational cost. This is an efficient solver specific capability. The work from [43] proposes to use multi-objective optimization for model generation. It focuses on the size of the generated models, which has been a constant concern on existing approaches. This feature and other algorithm dependent approaches would be interesting to be incorporated into a generic MS framework, adding value to the result. Many other model finding techniques developed outside of a modeling framework [44, 48, 47] could also be used as alternative model search implementations.

Approaches such as [29, 15] propose to integrate MDE/transformations and optimization techniques. The approach from [15] produces a model transformation, which can be further executed. The major contribution is not to provide a full translation between modeling and solver technical spaces, but to find the best rule execution sequence, from a set of existing rules. They also apply different optimization algorithms to choose the best transformation rules. The approach is extended in [14], concentrating on the quality of the solution and improving the choice of the optimization algorithms. They use HOTs (Higher Order Transformations) to infer information about the transformations and to choose the algorithms.

Finding the most appropriate rules is also the focus from [26], which applies combinatorial optimization techniques for model transformations. The main contribution is to find the transformation rules, based on an existing set of examples. They give special attention on bringing search based artifacts and algorithms, such as fitness functions, into an MDE environment. In [28], they present MotoE, which sees MT as a combinatorial optimization problem. It is one of the first works where the transformation is obtained from a set of examples. They use heuristic strategies to build the transformation. One important difference from rule based search approaches, such as ours, is that it does not intend to produce the solver specification from a given model transformation and the input models, but it searches for the input transformation itself. In [27], the authors present a solution adapted to model repair applied to a transformation scenario. The changes between models are expressed in terms of refactorings.

These approaches are relatively distant from ours on their main goals and design choices, because in our TAS approach the transformations are created individually, according to the developer specification, thus being more similar to rule-based specifications. In order to add optimization functions, it would be necessary to add specific transformations handling the input transformation language. Our approach loops over the set of the solutions provided by the solver, but without (so far) support to optimization.

Model relations and transformations as search

A number of studies have tackled establishing relations (also referred as links or relationships) between model elements and to use them in a large variety of scenarios. The utilization of these relationships for model transformations and synchronization can be coupled with search based solutions. Despite having different central goal, and some distinct research issues, the architecture and core facilities of such solutions can be related with a transformation as search framework.

The survey from [10] presents several studies about bidirectional model transformations. The work from [21] presents a classification of features of bidirectional transformations, though covering a larger scope, most of the approaches not involving model finding. We concentrate on the solutions that could be classified within a model generation context.

Different works have tackled synchronization issues. A number of incremental approaches [18, 5, 46] allow to update a target model by taking into account incremental changes on the source model. [41, 36, 45] handle synchronization as well, with special focus on model inconsistencies. These techniques inherit the deterministic behaviour of rule-based transformations. [20] proposes, based on abductive reasoning, to reverse an unidirectional transformation in order to provide synchronization of the source model with the previously obtained target model. In particular, it shares our ability to compute different alternative solutions through combinatorial logic inference. Here again, additional specifications are always required for the incremental/synchronization scenarios.

The JTL (Janus Transformation Language) [7] language supports non-bijective transformations and change propagation. The model transformations are translated into an ASP (Answer Set Programming) program. The search problem is generated from one source and one target models. The approach from [13] has the same core implementation of JTL, but extends it to support transformations between ADLs (Architecture Description Languages). It is one of the few approaches that have a detailed description of all the chain, and it is also based on the concept of a weaving model to set the links between source and target models, placing the work close to ours. However, it focuses on a star architecture and it provides only source-to-target model relations. It has metamodel independent translations through a generic bridge between technical spaces. They implemented Higher Order Transformations (HOTs) that generate the input and output bridges. However, the bridges need to be regenerated every time a source or target metamodel change. In our approach, we use only 2 reflexive transformations as bridges, one for input and another for output. This prevents from (re)generating multiple transformations that need to be managed. In addition, HOTs are often difficult to implement and error-prone.

In PTL (Prolog-based Transformation Language) [3], the authors translate a subset of ATL+OCL into Prolog, thus providing logical semantics for model transformations. Similar to our approach, it develops a translation from the modelings space to the solver space. They focus in a 1-to-1 transformation scenario, without describing how extensible the approach is.

Triple graph grammars (TGGs) [42] share the use of non-deterministic propagation mechanisms as foundations for their bidirectional and synchronization capabilities [16]. Their definition of a correspondence graph, though not grounded on basic modeling elements, is similar to our weaving and unification approach. Different specifications are suggested for the two scenarios as the generated language is further made deterministic through a set of sufficient static conditions. Additionally, the nature of triple graph rules restricts their use to one-and-one specifications.

Finally, the ECHO framework [34, 36] supports bidirectional model transformations. It provides a QVT-R implementation that is transformed into Alloy. Here the specifications are shared for both bidirectional and synchronization scenarios, but the approach is inherently restricted to the Alloy/SAT solver and one-and-one scenarios. It provides a complete translation chain and framework. As the authors stated, it is enumerative, though this is hidden from the user. [35] extends this work, providing a more detailed and complete implementation, handling both QVT-R and ATL. ECHO and the solutions that are implemented using solvers with monotonic behaviour circumvent the restriction of only extending a given model by re-generating the target model even for synchronization scenarios.

We provide a solver-independent specification of the bi-directional and synchronization problems in terms of model search. This means any specific problem that relates 2 or more models (synchronization, model repair, multi-directional transformations, or others) can be mapped into our TAS and then MS specification. As a consequence, ECHO's Alloy mapping, as any other approach from the previous paragraph, could be ported as a model search alternative implementation and would therefore be directly usable for any scenario. Additionally, the implementation of QVT-R could be achieved through a mapping to a corresponding weaving constrained metamodel. As drawback, application-specific aspects that can be related to the solver, or to the user interaction as well, are not specifically handled. This would require a per-approach study.

6. Conclusion

We presented a two-layer approach for implementing different kinds of model operations on terms of finite model generation techniques. The model operations, which may involve several input and output models, need to be translated into a model search problem, using only MDE artifacts and techniques. The model search specification is then translated into a solver-specific problem, which is translated back into the modeling world after resolution. This process is divided in a chain of steps, which are divided in solver-independent and solver-specific parts. The separation between the search problem from the actual solving means that depending on the solver, the domain of reachable solutions may be different. However, the way the problem is stated does not impose any restrictions on solutions.

We formalized and detailed the model search layer, which is a first-class operation independent from the solving back-end. We describe the chain of transformations needed to implement the process. This layer is the cornerstone for any other operation implementation. An example implementation of the solver-specific parts is then illustrated using the Alloy/SAT combination. This allowed to emphasize the problems that naturally arise in such implementations. A first set of experiments validated the applicability but also showed the limitations of boolean solvers for real engineering problems that include arithmetic computations or string manipulations. We believe that any model generation operation may be transformed into a model search problem and that our solver-agnostic approach will therefore ease the use and comparison of different back-ends on different types of problems. The solver-specific characteristics depend on the implemented solution.

We defined the concept of multi-level transformations, which are responsible for bridging between the modeling and the solver technical spaces. They are transformations implemented using reflection techniques, taking the metamodel and model as input, without explicitly referring to the elements of the input metamodels, allowing to realize the whole chain without any problem-dependent transformations. This is important to avoid developing one new transformation for each new input specification.

We then formalized and described the second layer, transformation as search, which allows to turn any model transformations and synchronization specifications into a model search specification. It is important to note that in transformation scenarios we have the source, target and weaving models translated into a single model search problem, by applying an unification strategy. To the best of our knowledge, the definitions of the unification strategy couple with weaving models has not been explored deeply in the literature. The implementation is completely independent from any solver and offers several advantages: multi-directional, incremental, single specifications for all scenarios, not limited to one-and-one operations. Our experiments showed that it may be particularly fitted for the synchronization scenario which is less impacted by the inherent combinatorial explosion of the approach.

Ongoing and future work include developing search chains that target different types of solvers to address specific computation needs (arithmetic, strings manipulation, etc.). This would allow to compare different solvers characteristics and to guide the choice of a given solver. Another future work is to support higher-level transformation language (such as QVT-R or ATL) to ease the writing of some types of specifications. Finally, we have not tackled in this paper the use of an optimization objective which may help the user in discriminating between potential solutions.

References

[1] *Eclipse OCL project: https://projects.eclipse.org/projects/modeling.mdt.ocl*, 2018.

[2] *MOS: https://gitlab.massidia.net/mos/software*, 2018.

[3] J. M. Almendros-Jimenez, L. Iribarne, J. Lopez-Fernandez, and A. Mora-Segura. PTL: A model transformation language based on logic programming. *Journal of Logical and Algebraic Methods in Programming*, 85(2):332 – 366, 2016.

[4] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9(1):69–86, 2010.

[5] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró. Incremental pattern matching in the Viatra model transformation system. In *proc. of 3rd International Workshop on Graph and*

Model Transformations, pages 25–32, Leipzig, Germany, 2008. ACM.

[6] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *proc. of International Conference on Automated Software Engineering*, Atlanta, USA, 2007.

[7] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: A bidirectional and change propagating transformation language. In *proc. of 3rd International Conference on Software Language Engineering*, pages 183–202, Eindhoven, The Netherlands, 2010.

[8] S. A. Cook. The complexity of theorem-proving procedures. In *proc. of ACM Symposium on Theory of Computing*, pages 151–158, Ohio, USA, 1971. ACM.

[9] A. Cunha, A. G. Garis, and D. Riesco. Translating between Alloy specifications and UML Class diagrams annotated with OCL. *Software and System Modeling*, 14(1):5–25, 2015.

[10] K. Czarnecki, J. N. Foster, Z. Hu, R.Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *proc. of International Conference on Model Transformations*, volume 5563, pages 260–283, Zurich, Switzerland, 2009.

[11] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45:621–645, 2006.

[12] *EMF: The Eclipse Modeling Framework : http://www.eclipse.org/modeling/emf/*, 2018.

[13] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. A Model-Driven approach to automate the propagation of changes among architecture description languages. *Software and Systems Modeling*, 11(1):29–53, 2012.

[14] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi. Model transformation modularization as a many-objective optimization problem. *IEEE Transactions on Software Engineering*, 43(11):1009–1032, 2017.

[15] M. Fleck, J. Troya, and M. Wimmer. Search-based model transformations. *Journal of Software: Evolution and Process*, 28(12):1081–1117, 2016.

[16] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and System Modeling*, 8(1):21–43, 2009.

[17] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computing Programming*, 69(1-3):27–34, 2007.

[18] D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In *proc. of Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 321–335, Genova, Italy, 2006.

[19] A. Hegedüs, A. Horváth, I. Ráth, M. C. Branco, and D. Varró. Quick fix generation for dsmls. In *proc. of IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 17–24, Pittsburgh, PA, USA, 2011.

[20] T. Hettel, M. Lawley, and K. Raymond. Towards model round-trip engineering: An abductive approach. In *proc. of International Conference on Model Transformations*, volume 5563 of *LNCS*, pages 100–115, Zurich, Switzerland, 2009.

[21] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. Feature-based classification of bidirectional transformation approaches. *Software and System Modeling*, 15(3):907–928, 2016.

[22] D. Jackson. Automating first-order relational logic. In *proc. of Foundations of Software Engineering*, pages 130–139, San Diego, California, USA, 2000.

[23] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19/20:503–581, 1994.

[24] F. Jouault and J. Bézivin. KM3: A DSL for metamodel specification. In *proc. of Formal Methods for Open Object-Based Distributed Systems,*, pages 171–185, Bologna, Italy, 2006.

[25] F. Jouault and I. Kurtev. Transforming models with ATL. In *proc. of MoDELS Satellite Events*, pages 128–138, Montego Bay, Jamaica, 2005.

[26] M. Kessentini, P. Langer, and M. Wimmer. Searching models, modeling search: On the synergies of sbse and mde. In *proc. of International Workshop on Combining Modelling and Search-Based Software Engineering*, pages 51–54, York, UK, 2013.

[27] M. Kessentini, U. Mansoor, M. Wimmer, A. Ouni, and K. Deb. Search-based detection of model level changes. *Empirical Software Engineering*, 22(2):670–715, April 2017.

[28] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. B. Omar. Search-based model transformation by example. *Software and Systems Modeling*, 11(2):209–226, May 2012.

[29] M. Kessentini, H. A. Sahraoui, and M. Boukadoum. Model transformation as an optimization problem. In *proc. of Model Driven Engineering Languages and Systems*, pages 159–173, Toulouse, France, 2008. Springer.

[30] M. Kleiner, M. Didonet Del Fabro, and P. Albert. Model search: Formalizing and automating constraint solving in MDE platforms. In *proc. of European Conference on Modelling Foundations and Applications*, pages 173–188, Paris, France, 2010.

[31] M. Kleiner, M. Didonet Del Fabro, and D. De Queiroz Santos. Transformation as search. In *proc. of European Conference on Modelling Foundations and Applications*, pages 54–69, Montpellier, France, 2013.

[32] T. Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[33] M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In *proc. of MoDELS Satellite Events*, pages 139–150, Montego Bay, Jamaica, 2005.

[34] N. Macedo and A. Cunha. Implementing QVT-R bidirectional model transformations using alloy. In *proc. of International Conference on Fundamental Approaches to Software Engineering*, volume 7793, pages 297–311, Rome, Italy, 2013.

[35] N. Macedo and A. Cunha. Least-change bidirectional model transformation with QVT-R and ATL. *Software and Systems Modeling*, 15(3):783–810, July 2016.

[36] N. Macedo, A. Cunha, and T. Guimarães. Exploring scenario exploration. In *proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 301–315, London, UK, 2015.

[37] N. Macedo, T. Guimarães, and A. Cunha. Model repair and transformation with Echo. In *proc. of International Conference on Automated Software Engineering*, pages 694–697, Silicon Valley, CA, USA, 2013.

[38] N. Macedo, J. Tiago, and A. Cunha. A feature-based classification of model repair approaches. *IEEE Trans. Software Eng.*, 43(7):615–640, 2017.

[39] S. Maoz, J. O. Ringert, and B. Rumpe. CD2Alloy: Class diagrams analysis using alloy revisited. In *proc. of Model Driven Engineering Languages and Systems*, pages 592–607, Wellington, New Zealand, 2011.

[40] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.1*, 2011.

[41] J. PinnaPuissant, R. Van Der Straeten, and T. Mens. Resolving model inconsistencies using automated regression planning. *Software and Systems Modeling*, 14(1):461–481, Feb 2015.

[42] A. Schürr and F. Klar. 15 years of triple graph grammars. In *proc. of the International Conference on Graph Transformations*, pages 411–425, Leicester, UK, 2008.

[43] O. Semeráth, A. Vörös, and D. Varró. Iterative and incremental model generation by logic solvers. In *proc. of Fundamental Approaches to Software Engineering*, pages 87–103, Eindhoven, The Netherlands, 2016.

[44] J. Slaney. FINDER: Finite domain enumerator system description. In *proc. of International Conference on Automated Deduction*, pages 798–801, Nancy, France, 1994.

[45] R. Van Der Straeten, J. Pinna Puissant, and T. Mens. Assessing the kodkod model finder for resolving model inconsistencies. In *proc. of European Conference on Modelling Foundations and Applications*, pages 69–84, Birmingham, UK, 2011.

[46] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Incremental model synchronization for efficient run-time monitoring. In *proc. of MoDELS Workshops*, volume 6002 of *LNCS*,

pages 124–139, Denver, Colorado, USA, 2009.

[47] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortez. Automated diagnosis of product-line configuration errors in feature models. In *proc. of Software Product Lines Conference, September 8-12*, Limerick, Ireland, 2008.

[48] J. Zhang and H. Zhang. System description generating models by sem. In *proc. of International Conference on Automated Deduction, July 30 - August 3*, pages 308–312, New Brunswick, NJ, USA, 1996.

Appendix A. Model Search generated specification

We list below the complete Alloy specification generated for the class diagram (CD) Model Search problem. It contains the initial instance (*module RootModel*), the metamodel specification (*module CD*) and additional constraints generated from OCL (*module generation*).

```
module RootModel
open CD

one sig Class0 extends Class  {
}
one sig Package0 extends Package  {
}
fact  {
 (Class0 in (Package0.classes))
}
fact  {
 ((Package0.name) = "Top")
}
fact  {
 (Package0 in (Class0.package))
}
fact  {
 ((Class0.name) = 0)
}

module CD
open util/integer
open util/integer

sig Package  {
 name: one String
, classes: set Class
}
sig Class  {
 attributes: set Attribute
, package: one Package
, children: set Class
, name: one Int
, parent: lone Class
}
sig Attribute  {
 name: one Int
, owner: one Class
}
fact structuralPropertiesForAttributeowner  {
 (owner = (~attributes))
}
fact structuralPropertiesForClasschildren  {
 (children = (~parent))
}
fact structuralPropertiesForClassattributes  {
 (((Class <: attributes) in Class lone -> Attribute) &&
 (attributes = (~owner)))
}
fact structuralPropertiesForClassparent  {
 (parent = (~children))
}
fact structuralPropertiesForPackageclasses  {
 (((Package <: classes) in Package lone -> Class) &&
 (classes = (~package)))
}
fact structuralPropertiesForClasspackage  {
 (package = (~classes))
}
```

```
module generation
open util/integer
open CD
open util/boolean

fact nonCircularInheritance  {
 (all self: Class | (((#(self.parent)) = 1) => (not(self in
 ((self.parent).(*parent))))))
}
fact differentNames  {
 (all self: Class | ((#{c: (Class - self) |
 ((c.name) = (self.name))}) = 0))
}
fact parentName  {
 (all self: Class | (((#(self.parent)) = 1) =>
 (((self.parent).name) = ((self.name).minus[1]))))
}
fact attributeName  {
 (all self: Attribute | ((self.name) = ((self.owner).name)))
}
fact maxOneChild  {
 (all self: Class | ((#(self.children)) < 2))
}
fact hasOneAttribute  {
 (all self: Class | ((#(self.attributes)) = 1))
}
fact allButTopHaveAParent  {
 (all self: Class | ((not((self.name) = 0)) =>
 ((#(self.parent)) = 1)))  }
```

Appendix B. Transformation as Search generated specification

We list below the Alloy code of the weaving metamodel (*module Class4Relational*) and the constraints from the OCL specification (*module C4TASMM*) with the TAS specification. The left (*Class*) and right (*Relational*) metamodels, as well as the Root Model specifications are available for download in the prototype site.

```
module Class4Relational
open util/boolean
open Class
open Relational

sig ClassAndTable extends WLink  {
 table: one Table
, class: one Class
}
sig DataTypeAndType extends WLink  {
 type: one Type
, datatype: one DataType
}
abstract sig WLink  {
}
sig AttributeAndColumn extends WLink  {
 attribute: one Attribute
, column: one Column
}
sig AttributeAndTable extends WLink  {
 table: one Table
, valueIsForeignKey: one Bool
, valueColumn: one Column
, sourceColumn: one Column
, attribute: one Attribute
}

module C4TASMM
open util/boolean
open Class4Relational
open util/integer
open Class
open Relational

fact ColumnWithDifferentName  {
 (all self: Column | (all c2: ((self.owner).col) | ((c2 in Column)
 and ((not(self = c2)) => (not((self.name) = (c2.name)))))))
}
```

```
fact TableColumn  {
 (all self: Attribute | (((#(self.(~(AttributeAndColumn <:
 attribute)))).plus[(#(self.(~(AttributeAndTable <: attribute))))])
 = 1))
}
fact AttributeAndTableNames  {
 (all self: AttributeAndTable | (((self.attribute).name) =
 ((self.table).name)))
}
fact FKNotMultivaluedOrClass  {
 (all self: Column | (((#(self.foreignKeyTo)) = 1) =>
 ((((#(self.(~(AttributeAndColumn <: column)))) = 1) and
 ((((self.(~(AttributeAndColumn <:
 column))).attribute).multiValued) = False)) or
 (((#(self.(~(AttributeAndTable <:
 sourceColumn)))).plus[(#(self.(~(AttributeAndTable <:
 valueColumn))))]) = 1))))
}
fact TablesWithPrimaryKey  {
 (all self: Table | ((self.primaryKey) in (self.col)))
}
fact ForeignKeyColumn  {
 (all self: Column | (((((#(self.foreignKeyTo)) = 0) and
 (not(((self.owner).primaryKey) = self))) and
 ((#(self.(~(AttributeAndTable <: valueColumn)))) = 0)) =>
 ((#(self.(~(AttributeAndColumn <: column)))) = 1)))
}
fact WeavedColumnForeignKey  {
 (all self: AttributeAndColumn | ((((self.attribute).type) in
 Class) => (((self.column).foreignKeyTo) =
 (((((self.attribute).type) :> Class).(~(ClassAndTable <:
 class))).table))))
}
fact ClassAndTableNames  {
 (all self: ClassAndTable | (((self.class).name) =
 ((self.table).name)))
}
fact WeavedSourceForeignKey  {
 (all self: AttributeAndTable | (((self.sourceColumn).foreignKeyTo)
 = ((((self.attribute).owner).(~(ClassAndTable <: class))).table)))
}
fact ForeignKeyValueColumn  {
 (all self: AttributeAndTable | (((self.valueIsForeignKey) = True)
 => (((self.valueColumn).foreignKeyTo) = (((((self.attribute).type)
 :> Class).(~(ClassAndTable <: class))).table))))
}
fact NotMultivaluedAndWeavedType  {
 (all self: AttributeAndColumn | (((((self.attribute).type) in
 DataType) and (((self.attribute).multiValued) = False)) =>
 (((self.column).type) = (((((self.attribute).type) :>
 DataType).(~(DataTypeAndType <: datatype))).type))))
}
fact  {
 (all self: Class | (all a1: (self.attr) | ((a1 in Attribute) and
 (all a2: ((a1.owner).attr) | ((a2 in Attribute) and ((not(a1 =
 a2)) => (not((a1.name) = (a2.name)))))))))
}
fact DataTypeAndTypeNames  {
 (all self: DataTypeAndType | (((self.datatype).name) =
 ((self.type).name)))
}
fact ValueSourceColumnDifferent  {
 (all self: AttributeAndTable | (not((self.sourceColumn) =
 (self.valueColumn))))
}
fact ClassTable  {
 (all self: Class | ((#(self.(~(ClassAndTable <: class)))) = 1))
}
fact DataTypeType  {
 (all self: DataType | ((#(self.(~(DataTypeAndType <: datatype))))
 = 1))
}
fact AttributeAndColumnNames  {
 (all self: AttributeAndColumn | (((self.attribute).name) =
 ((self.column).name)))
}
fact WeavedSourceBelongsWeavedTable  {
 (all self: AttributeAndTable | (((self.sourceColumn).owner) =
 (self.table)))
}
fact MultiValuedWithWeavedOwner  {
 (all self: AttributeAndColumn | ((self.column) in
 (((((self.attribute).owner).(~(ClassAndTable <:
 class))).table).col)))
}
fact ValueColumnForeignKey  {
 (all self: AttributeAndTable | ((((self.attribute).type) in Class)
  => ((self.valueIsForeignKey) = True) else
  ((self.valueIsForeignKey) = False)))
}
fact WeavedValueBelongsWeavedTable  {
 (all self: AttributeAndTable | (((self.valueColumn).owner) =
 (self.table)))
}
fact AttributeColumn  {
 (all self: Attribute | (((self.multiValued) = False) =>
 ((#(self.(~(AttributeAndColumn <: attribute)))) = 1)))
}
fact TypeDataType  {
 (all self: Type | ((#(self.(~(DataTypeAndType <: type)))) = 1))
}
fact WeavedSourceName  {
 (all self: AttributeAndTable | (((self.sourceColumn).name) =
 "source"))
}
fact ValueColumnsWeavedType  {
 (all self: AttributeAndTable | (((self.valueIsForeignKey) = False)
 => (((#((self.valueColumn).foreignKeyTo)) = 0) and
 (((self.valueColumn).type) = (((((self.attribute).type) :>
 DataType).(~(DataTypeAndType <: datatype))).type)))))
}
fact PrimaryKeysIntegers  {
 (all self: Table | (((#(self.primaryKey)) = 1) =>
 ((((self.primaryKey).type).name) = "Integer")))
}
fact PrimaryKeyIsNotForeignKey  {
 (all self: Table | ((#((self.primaryKey).foreignKeyTo)) = 0))
}
fact TableAttribute  {
 (all self: Table | (((#(self.(~(AttributeAndTable <:
 table)))).plus[(#(self.(~(ClassAndTable <: table))))]) = 1))
}
fact WeavedValueName  {
 (all self: AttributeAndTable | (((self.valueColumn).name) =
 "target"))
}
fact AttributeTable  {
 (all self: Attribute | (((self.multiValued) = True) =>
 ((#(self.(~(AttributeAndTable <: attribute)))) = 1)))
}
fact Tablenames  {
 (all self: Table | (((self.primaryKey).name) = "ID"))
}
fact ForeignKeyTypeEqualsPrimaryKeyType  {
 (all self: Column | (((#(self.foreignKeyTo)) = 1) =>
 (((#((self.foreignKeyTo).primaryKey)) = 1) and ((self.type) =
 (((self.foreignKeyTo).primaryKey).type)))))
}
```