Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Model Finding in the EMF Ecosystem

Jesús Sánchez Cuadrado[a]   Martin Gogolla[b]

a. University of Murcia (Spain)

b. University of Bremen (Germany)

Abstract The EMF framework is the main meta-modelling framework used nowadays. It has a rich ecosystem of plug-ins and tools built with and for it, including the option of enriching meta-models with OCL constraints. However, the EMF ecosystem lacks usable model finding approaches. Given a meta-model, a model finder automatically searches for models that satisfy a given set of formulas (e.g., OCL constraints). This feature can be used for a number of purposes, including model verification and model synthesis.

In this paper, we present an approach to support model finding in the EMF ecosystem that is designed to realize several scenarios including model consistency, example generation, partial solution completion and scrolling. Moreover, it allows several OCL variants to be plugged-in via an intermediate representation. This approach has been realized in a tool called EFinder. We have assessed the usability of the approach by implementing three advanced application scenarios and evaluated its verification capabilities by analyzing OCL constraints from an external OCL dataset containing about 300 valid EMF/OCL specifications. Our model finder is able to process about 65% of these EMF/OCL models.

Keywords Model, Meta-model, EMF model, Model validation and verification, Constraint solving, Model finding

1 Introduction

The Eclipse Modeling Framework (EMF) is nowadays one of the most commonly used meta-modeling frameworks, with a rich ecosystem of tools and extensions built around it. This includes language workbenches like Xtext and Sirius, model transformation and model management languages like ATL, Epsilon, and QVT Operational. EMF also features an implementation of the Object Constraint Language (OCL), allowing the developer to enrich EMF meta-models with OCL constraints, to evaluate the conformance of models and to validate models [Ecl17]. Moreover, many EMF-based tools make use of this OCL implementation or implement its own OCL variant, in order to express constraints or to use OCL as a navigation language (e.g., ATL and Sirius).

Given the first-order logic like nature of OCL, several works have put forward techniques to perform model finding for a given meta-model over OCL constraints (e.g., [KG12, GBCC12, SSB19]). Taking advantage of recent progress in SAT solving (for boolean satisfiability problems), a model finder takes a set of constraints written in some high-level language like OCL and some high-level model (in UML terms called a class diagram, or an Ecore meta-model in EMF), translates it to a SAT problem and applies a SAT solver to find a model (in UML terms called an object diagram) satisfying the constraints for the given meta-model. This technique is effective in many scenarios including model validation [GHD18] and transformation verification [CGdL17a].

However, despite the practical importance of the EMF framework in the community, model finding is generally not included in mainstream EMF-based tools. The reason could be that there is no working and reliable model finder that *seamlessly integrates with EMF and different OCL variants*. In this paper, we propose a framework that brings model finding capabilities into the EMF ecosystem. The framework aims to satisfy the following requirements:

1. It must provide robust model finding based on constraints written in OCL. In addition to basic satisfiability checking, we aim at providing support for additional features currently not supported in existing EMF finders such as partial model completion (i.e., the ability to extend an existing model to satisfy the given constraints) and scrolling (i.e., to retrieve all valid models).

2. It must integrate seamlessly with EMF, which means it must naturally process EMF meta-models, i.e., consume and output EMF models using regular EMF resources.

3. It must cover a wide range of OCL features.

4. It must support several variants of OCL, and new variants should be easily supported in the future.

To achieve these goals, we have implemented a flexible model finding architecture based on a configurable and extensible intermediate OCL representation. As a model finding backend we apply the USE Model Validator [KG12, GHD18] (USE MV), which is a robust model finder for UML class diagrams and OCL constraints. We have implemented translations from EMF/OCL [EOT19], AQL [AT19] and ATL [JABK08] and enhanced our backend with internal transformations that provide support for OCL features like tuples, iteration and recursive operations which are not supported natively by USE MV. Moreover, in addition to conceptual limitations there are technogical issues which has been addressed in order to increase the usefulness of our approach (e.g., package flattening, renaming of reserved keywords, etc.). This approach has been realized in a tool, named EFinder, which is available for download as an additional contribution. We illustrate the applicability of our proposal with three scenarios that extend EMF with model finding, namely: model verification, automatic construction of examples to demonstrate a graphical notation and cross-artifact verification. Moreover, we have evaluated the expressiveness of EFinder against a third-party OCL dataset.
Organization. The paper is organised as follows. Section 2 introduces a running example and the architecture of our framework. The technical contribution is presented in Sect. 3, which discusses technical details about the translation process. Then, Sect. 4 further shows the usefulness of our approach by illustrating three application scenarios. Section 5 describes the tool and Sect. 6 reports on the evaluation results. Finally, Sect. 7 presents related work, and Sect. 8 summarizes the results and future work.

2 Overview

This contribution is motivated by the need for providing the EMF ecosystem with a robust model finder to enable automatic reasoning over EMF models. This section motivates and introduces model finding with a running example and presents the architecture of our approach.

2.1 Running example

As a running example, let us consider a subset of the relational data model, in other words a subset of SQL. Fig. 1 shows its Ecore meta-model and Listing 1 contains its associated invariants written in the *Complete OCL* variant[1]. A table, also called a Rel[ational] Schema, possesses attributes that are categorized as key or non-key attributes (isKey). Each attribute is typed through a datatype. A table can be populated with a number of rows that possess components for the attributes, technically realized by attribute map objects that are typed through attributes and that in turn refer to value objects.

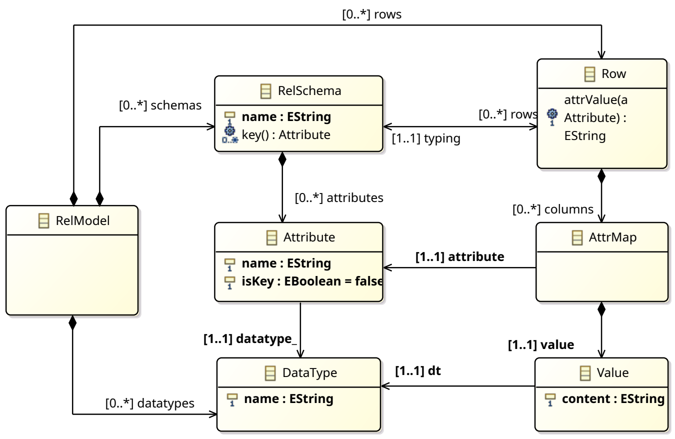
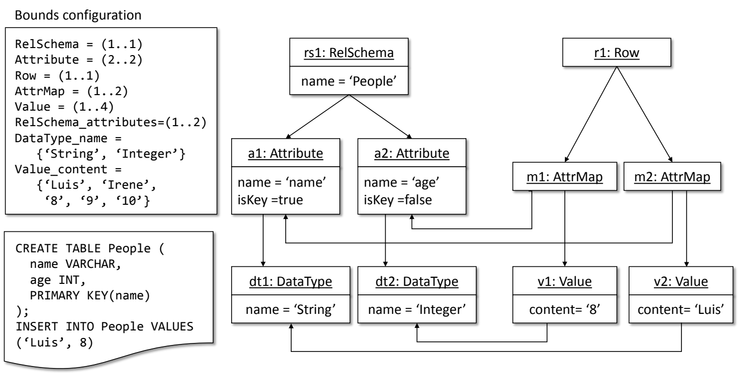


Figure 1 – Ecore meta-model for relational schema.

```
import 'relschema.ecore'

context RelSchema inv uniqueAttributeNames:
  self.attributes−>forAll(a1, a2 | a1.name = a2.name implies a1 = a2)

context RelSchema inv relSchemaKeyNotEmpty:
  self.key()−>notEmpty()

context Row inv keyMapUnique:
  Row.allInstances()−>forAll(self2 |
     self<>self2 and self.typing = self2.typing implies
         self.typing.key()−>exists(ka | self.attrValue(ka) <> self2.attrValue(ka)))

context Row inv hasAttrMapForAllAttr:
  self.typing.attributes−>forAll(aRS| self.columns.attribute−>one(aAM| aRS.name=aAM.name)) and
  self.columns.attribute−>forAll(aAM| typing.attributes−>one(aRS| aRS.name=aAM.name)) and
  self.columns−>size() = relSchema.attributes−>size()
```

Listing 1 – OCL constraints.

[1]It is an OCL dialect which provides a Xtext-based editor and compiles to Pivot OCL [EOT19]



Bounds configuration

```
RelSchema = (1..1)
Attribute = (2..2)
Row = (1..1)
AttrMap = (1..2)
Value = (1..4)
RelSchema_attributes=(1..2)
DataType_name =
   {‘String’, ‘Integer’}
Value_content =
   {‘Luis’, ‘Irene’,
    ‘8’, ‘9’, ‘10’}
```

```
CREATE TABLE People (
  name VARCHAR,
  age INT,
  PRIMARY KEY(name)
);
INSERT INTO People VALUES
(‘Luis’, 8)
```

Figure 2 – Witness model generated by the model consistency scenario.

Using our model finder we are interested in providing support to the following features in the EMF ecosystem.

Model Consistency checks whether the model can be instantiated by at least one object model given a finite search space, i.e., proving that the meta-model multiplicities together with the OCL invariants are not contradictory. In this case, a witness model is generated proving that such a model exists. A model finder could instantiate a model similar to the one shown in Fig. 2 (for the sake of clarity, the corresponding SQL statements that would generate an equivalent schema and database state are shown in the lower left as a note). In this case, the model has a single RelSchema with two attributes, for which there is a single row. Typically, a model finder requires a so-called configuration where obligatory upper bounds for the number of objects per class (e.g., exactly one Row object), optional attribute values (e.g., ‘String, ‘Integer’ for datatype names) and optional upper bounds for the number of links per reference (e.g., up to 2 links, i.e., RelSchema.attributes->size) are stated. On the basis of such a configuration, the object model is constructed, and, by this, the consistency of the stated multiplicities from the class model together with the explicit OCL invariants is proved.

Partial Solution Completion assumes a partially described object model is present which may not (yet) satisfy the model invariants; then a model finder would try to find a completion in terms of objects, references and attribute values such that a valid object model satisfying all constraints is generated. In the example, if the object v1:Value would not exist, the model finder would complete the model, adding at least one object, filling the attributes with some valid values and establishing appropriate links for the object v1:Value.

Scrolling refers to the capability of generating *all* possible models, within the given bounds, which conform to the meta-model and satisfy its associated OCL invariants. This is particularly interesting in combination with approaches like “Partitioning with Classifying Terms” [HGBV18] to generate many example models according to some criteria, e.g., achieving *diverse* object models that show considerable differences.

2.2 Architecture

The design of our framework intends to satisfy the requirements stated in the introduction and to implement the features described in the previous section. Fig. 3

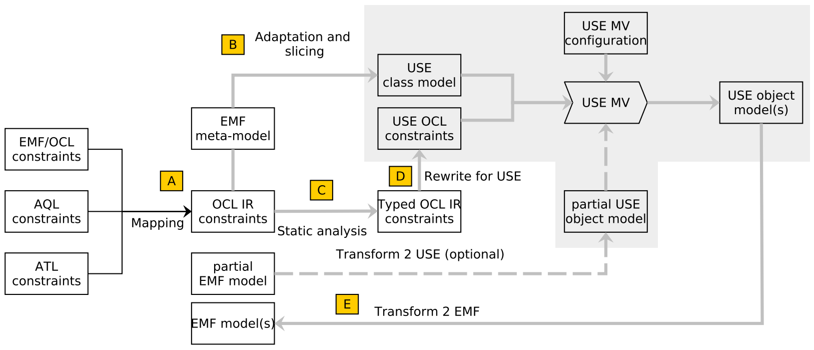


Figure 3 – Architecture of EFinder.

shows its architecture. The rationale of this architecture is to allow any EMF-based implementation of OCL to profit from the framework easily and to allow its integration in EMF-based tools. The framework is based on several components, annotated in the figure: (A) an OCL intermediate representation (IR), (B) a meta-model slicer, (C) a back-end. The back-end is intended to compile the OCL IR to some off-the-shelf contraint solver or another model finder. In particular, we currently make use of USE Model Validator (USE MV), which is a mature and robust OCL model finder. However, USE MV does not interact well with other model development ecosystems. EFinder's backend applies a number of transformations and algorithms to make it compatible with EMF.

We aim at facilitating tool developers in integrating a model finder into their tooling, since model finders may make the developed tools much more powerful. Hence, a key element in the framework design is the desire to provide support for several OCL variants and query languages compatible with EMF. To achieve this we propose to use an intermediate representation plus dedicated transformations which normalize certain details of a given dialect so that it is possible to treat it generically.

Our OCL IR is based on a simple abstract syntax (AST) which is intended to cover the constructs appearing in the existing OCL implementations in EMF. Given an OCL implementation, a mapping between its AST and the IR's AST must be developed (label A). So far, we have implemented mappings for EMF/OCL, ATL and AQL, and the implementations for other dialects is expected to be relatively simple since most elements are common between OCL variants.

The OCL IR can be optionally type-checked (label C). This step is recommended to gather additional information from the OCL text, which is later used to increase the quality and performance of the translation to the backend. The existence of a specific type checker for our OCL IR allows us to handle untyped variants like ATL, SimpleOCL and EOL. In the current implementation the type checking is performed using AnATLyzer's facilities, but we want to built a specific type checker in the future. In order to improve the quality of the type checking it is recommended to configure the system with a standard library definition matching the one of the source OCL variant. This is done with a dedicated fluent API. It is also possible to define analysis extensions to override some default behaviour which maybe different in a given variant. For instance, EMF/OCL has an operation to match regular expressions (matches). The type checker is configured with a definition of the EMF/OCL standard library which contains the definition (matches(String) : String).

The other key element of our approach are transformations applied to adapt the IR to the backend features and to improve the performance (labels B, D and E). These transformations are described in the next section.

Using this architecture we aim at enabling a number of advanced modelling scenarios for EMF models. We foresee applications related to (meta)-model verification, verification of model transformations, automatic generation of examples in different scenarios (e.g., to support live modelling), model-based testing, etc. To demostrate the feasibility of such scenarios we have build prototypes which are presented in Sect. 4. Nevertheless, our aim is to encourage the community to use our tool to build other types of applications.

3 Transforming OCL for model finding

Our approach is based on applying a series of transformations starting from a source OCL dialect, which is mapped to a IR over which in-place transformations are applied and finally a mapping to a backend. This section describes these transformations.

3.1 Meta-model slicing

Our architecture contains a dedicated phase for *meta-model slicing*. In this phase we analyse which meta-model elements are used by the OCL text to slice the original meta-model and reduce its size. To this end, the typed version of the OCL IR is traversed, looking for ocurrences of meta-classes (e.g., as in Row.allInstances) and accesses to structural features (e.g., self.attributes). Meta-model elements that are not encountered are discarded when constructing the sliced meta-model. This is useful to reduce the search space in those scenarios in which it is acceptable to discard parts of the meta-model. In the example, to prove that the constraints are satisfiable the model finder does not require class RelModel nor features like RelSchema.name. Please note that if the aimed functionality is, for instance, 'Partial Solution Completion' we may use the original meta-model.

3.2 Translation to the backend

In our current implementation we use USE Model Validator as a model finding backend. Thus, the OCL IR is rewritten in order to match the USE Model Validator features and structure. If it is not possible to perform a "good enough" translation to USE (e.g., one that does make it fail internally), the process is aborted. This step allows us to show to the user a precise description of why the constraint could not be evaluated (e.g., due to unsupported features), instead of silently failing with an internal exception. Finally, we generate code in the USE format for the meta-model and the constraints, and feed dynamically the model finder.

For the example, an excerpt of the translation to USE is shown in the following listing. First, classes required for the verification are translated to the USE format. Then, operations and attributes defined at the meta-model level (e.g., key() and isKey) are translated and merged. Please note that the system takes care of keyword clashes (e.g., attributes is renamed to attributes_). Each invariant is then translated to USE under the constraints section.

```
model relschema
```

```
class RelSchema
attributes
  name : String
operations
  key() : Set(Attribute) = self.attributes_->select(a| a.isKey)
end

class Attribute
attributes
  isKey : Boolean
  name : String
end

composition RelSchema_attributes between
  RelSchema[0..1] role RelSchema_attributes_source
  Attribute[*] role attributes_
end
...

constraints

context self : RelSchema inv RelSchema_2:
  self.attributes_->forAll(a1, a2| a1.name = a2.name) implies a1 = a2)
...
```

In addition, in this step we also adapt the Ecore meta-model to fit the USE class model features. In particular, one limitation of USE is that it does not support multiple class diagrams for the same OCL text. To support constraints which are typed against more than one Ecore meta-model we merge all meta-models into one, performing the required rewritings to avoid name clashes.

3.3 Integration of EMF and USE models

If the constraints are satisfiable, the result is an example model represented using data structures from USE, which needs to be translated back to the EMF side. This translation must take into account some of the transformations done in the previous rewritings. Moreover, there is the possibility of giving an existing EMF model as input. This model is translated at runtime into data structures from USE in order to realize the 'Partial Model Completion' scenario. This is a translation of object models, which is parameterized with the mapping performed by the previous translation in order to keep both models consistent.

3.4 Improving OCL coverage

This section presents some of the advanced features provided by EFinder in order to improve its coverage in terms of OCL features amenable to model finding and to bridge semantic differences between Ecore meta-models and USE meta-models.

Only one root. In EMF there exists the convention that a model should have one root element, which acts as a container (directly or indirectly) for the rest of the elements. In the example, RelModel plays such a role. As part of our translation there is an option to generate a constraint which enforces that all elements belong to a single root container (e.g., to avoid having a model with a RelModel instance and a RelSchema instance as roots). The following listing shows an excerpt of this constraint.

```
context RelModel inv oneModel:
  RelModel.allInstances()->size() = 1 and
  RelSchema.allInstances()->forAll(r | not r.oclContainer().oclIsUndefined()) and
  ... -- same pattern for Attribute, Row, DataType, AttrMap and Value

```

```
context RelSchema operation oclContainer() : OclAny =
  RelModel.allInstances()−>select(r | r.schemas−>includes(self))−>any(true)
```

Three-valued logic. The OCL standard, and USE in particular, uses a three-valued logic (e.g., there are three truth values: true, false and undefined). However, in EMF primitive values cannot be identical to OclUndefined, therefore we establish a constraint to make USE behave like a two-valued logic for primitive attributes. The following listing shows an example of such a constraint.

```
−− Three−valued logic mapping
context Attribute inv tvl:
  not self.isKey.isUndefined()
```

Ranges. In OCL it is possible to describe a collection of primitive values using a range, for example, Set {1..5}. This is supported in EMF/OCL, but it is not supported yet in the USE MV. In many cases it is possible to unfold the range into its components to generate all collection elements explicitly (e.g., Set {1, 2, 3, 4, 5}).

Recursion. The USE MV does not support recursive operations. For instance, let us suppose that our Relational meta-model additionally supports inheritance. The following operation and the associated constraint cannot be verified.

```
context RelSchema
operation allAttributes() : Set(Attribute) =
  if self.parent.oclIsUndefined() then Set { }
  else self.attributes−>union(self.parent.allAttributes()) endif

context RelSchema inv uniqueAttributeNames_Inheritance:
  self.allAttributes()−>forAll(a1, a2 | a1.name=a2.name implies a1=a2)
```

Our approach to deal with this issue is based on unfolding a recursive operation up to a finite number of steps. We perform the unfolding by copying the original operation n times, so that there are $n+1$ versions of the operation. Then, each version of the operation is rewritten so that the recursive call site does not invoke the original operation, but the next copy of the operation. The last operation in the sequence just returns OclUndefined as a bottom value (i.e., to indicate an evaluation error). Listing 2 shows a sketch of this procedure. It takes the desired number of unfoldings (N) and the piece of abstract syntax corresponding to the recursive operation (OP). There are two helper functions, callSites which returns the set of recursive call sites (i.e., a set of abstract syntax elements representing operation calls that invoke Op) and copy which returns a deep copy of the given abstract syntax element.

```
N = Number of unfoldings
OP = Original operation

$OP_0$ = OP
for i = 1 to N
  $CS_{i-1}$ = callSites($OP_{i-1}$)
  foreach cs in $CS_{i-1}$
    cs.operationName = OP.operationName + "_" + i
  end

  $OP_i$ = copy(OP)
  $OP_i$.operationName = OP.operationName + "_" + i
end

$OP_N$.body = OclUndefined
```

Listing 2 – Sketch of the unfolding algorithm.

In this way, the allAttributes operation is unfolded for $n = 2$ as follows:

```
context RelSchema
operation allAttributes() : Set(Attribute) =
  if self.parent.oclIsUndefined() then Set { }
  else self.attributes−>union(self.parent.allAttributes_1()) endif

operation allAttributes_1() : Set(Attribute) =
  if self.parent.oclIsUndefined() then Set { }
  else self.attributes−>union(self.parent.allAttributes_2()) endif

operation allAttributes_2() : Set(Attribute) = OclUndefined
```

Iterate. The iterate operation is also not supported in the USE MV. We use a similar approach as with recursive operations. For each occurrence of iterate we generate a specific recursive operation which implements it, and then apply the unfolding explained above.

Tuples. OCL supports anonymous tuples, which allows temporary objects to be created. For illustration purposes, let us consider the following use of tuples to describe elements of a relational model: we want to design two schemas (Person and Pet) with key attributes, named id, and nickname, respectively.

```
context RelModel
def: example() : Set(Tuple (rel : String, keyName : String)) =
  Set { Tuple { rel = 'Person', keyName = 'id' },
        Tuple { rel = 'Pet', keyName = 'nickname' } }

context RelModel inv withExample:
  self.example()−>forAll(e | self.schemas−>exists(s |
     s.name = e.rel and s.key()−>one(a | a.name = e.keyName)))
```

We have implemented a dedicated rewriting, at the OCL IR level, to support tuples in model finders like the USE MV which does not have support for them. The underlying idea is to automatically extend the source meta-model with one class per tuple type. In this case, we name this class RelKeyNameTuple. Then, we need to force the model finder to create the corresponding objects using the strategy exemplified in the following listing. First, we need to replace each tuple literal with some OCL expression to retrieve an object whose features match the values originally assigned to the tuple (lines 3–4). There is however a caveat. A valid result would be that any does not find an object, returning OclUndefined. Hence, we need to add an additional constraint to avoid this. Our current approach is to identify in which locations of the OCL text a tuple is accessed (e.g., s.name, line 9), and find the "boolean location" in which it is possible to insert the constraint that a tuple cannot be OclUndefined (line 8 in the example).

```
1 context RelModel
2 def: example() : Set(RelKeyNameTuple) =
3   Set{RelKeyNameTuple.allInstances()−>any(rel='Person',keyName='id'),
4       RelKeyNameTuple.allInstances()−>any(rel='Pet',keyName='nickname') }
5
6 context RelModel inv withExample:
7   self.example()−>forAll(e | self.schemas−>exists(s |
8      not e.oclIsUndefined() and −− Constrain the tuple
9      s.name = e.rel and s.key()−>one(a | a.name = e.keyName)))
```

The transformations described in this section provides the means to satisfy the requirements stated in the introductory section. The translation to and from USE models and class diagrams is completely transparent to the user, so that both the input and the output of the model finding process use EMF resources, which can be seamlessly manipulated in memory or stored on disk. The coverage of OCL features depends on the support of the USE MV, but our approach based on OCL IR rewritings allows us to circumvent some of its limitations and to increase the coverage (see Sect. 6).

In addition, the OCL IR allows OCL variants to be easily plugged-in and to profit from the rest of the infrastructure (e.g., meta-model slices, IR rewritings, conversions to USE). We currently support three OCL variants, but the implementation of new variants in the future is expected to be relatively straightforward. Finally, we also take advantage of our OCL IR transformations to analyse the input OCL text to detect which features are not supported and to report failures properly.

4 Applications

This section describes three scenarios in which model finding can be used to prove properties and to generate example models, and in which the EFinder architecture provides integration with different EMF technologies.

4.1 Model analysis and exploration

Models and meta-models are key elements in Model-Driven Engineering. Meta-models need to be well engineered in order to allow modellers construct useful and correct models. In this setting OCL constraints play the important role of adding additional semantics to a meta-model. This scenario illustrates the use of EFinder as a tool to analyze and explore models, meta-models and OCL constraints. This brings to EMF modellers part of the functionality already available in USE [GHD18] which was briefly introduced in Sect. 2.

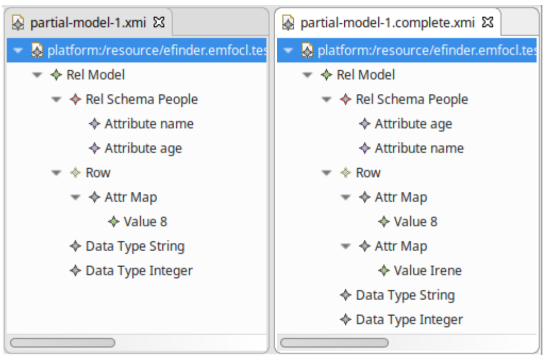
Model Consistency. This functionality is realized by letting EFinder generate an example model which satisfies the given constraints. If such a model exists within the given bounds (i.e., within the configuration given to EFinder), then the meta-model and the constraints are consistent. This has already been illustrated in Fig. 2. We currently support two approaches for handling the bounds: a) the user may provide an external file or direct annotations in the OCL text to give explicit values to the minimum and maximum number of objects, and also for integer ranges and for explicit string values, or b) the system automatically tries to determine the proper bounds automatically by searching for a solution starting with some small bounds, e.g., (1..3), up to a maximum, with a configurable maximum bound e.g., (1..8).

Partial Solution Completion. Given an EMF Resource, which already contains a set of model elements, this functionality allows the developer to derive a new version of the model in which all the constraints are satisfied. Figure 4 shows (a) a model (left) that violates the hasAttrMapForAllAttr constraint, and (b) the new version of the model (right) in which the constraint is now satisfied by introducing an AttrMap object.

Scrolling. In the previous two scenarios EFinder outputs one example model if the constraints are satisfiable. However, there might be other example models available. We have integrated the *scrolling* functionality of the USE MV to allow the user to return the rest of the solutions iteratively.

4.2 Sirius previewer

Sirius is a tool to implement graphical editors for EMF models in a declarative way by creating an *odesign* model, which defines (among other elements) a mapping between the language abstract syntax (its Ecore meta-model) and a concrete syntax defined by the Sirius graphical meta-model. For instance, Fig. 5 (label A) shows the definition of a graphical editor for a simple Statecharts meta-model which contains States and



```
partial-model-1.xmi
platform:/resource/efinder.emfocl.te
  Rel Model
    Rel Schema People
      Attribute name
      Attribute age
    Row
      Attr Map
        Value 8
    Data Type String
    Data Type Integer
```

```
partial-model-1.complete.xmi
platform:/resource/efinder.emfocl.te
  Rel Model
    Rel Schema People
      Attribute age
      Attribute name
    Row
      Attr Map
        Value 8
      Attr Map
        Value Irene
    Data Type String
    Data Type Integer
```

Figure 4 – Input model for model completion (left) and completed model (right).

Transitions. For each notation element that we want to show in the editor, we need to create a mapping to its corresponding meta-model element. For instance, to represent initial states (using a black circle) a mapping to the State meta-class is defined, taking into account the precondition that this mapping only holds when isInitial = true. In the case of edges, like Transition, we also need to specify the possible source and target notation elements (depicted with a normal line end and a line with an arrow, respectively).

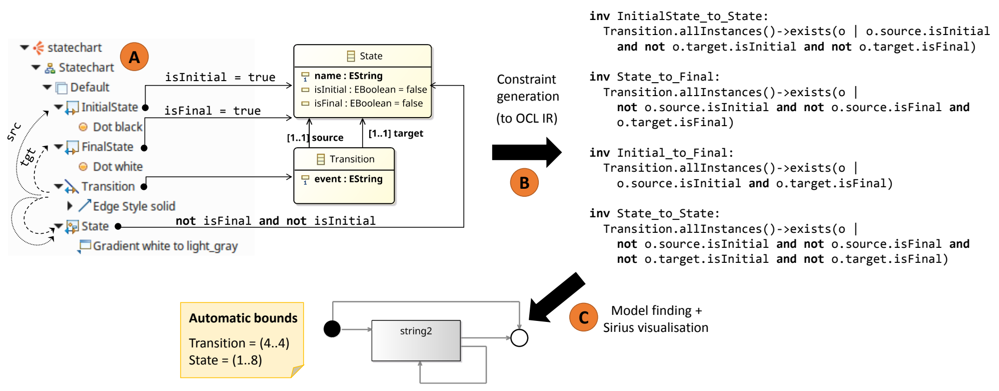
Developing an editor in Sirius typically implies the following cycle of development steps: define a mapping for a meta-model element, add some graphical properties (color, shape, etc.), create a new model or modify an existing one to instantiate some objects (i.e., create a new XMI file using the regular tree editor), update some properties of interest, and then open a new editor to visualize the example model. If the graphical rendering is not as expected, the editor implementation must be changed. This process is time consuming and could be improved.

To automate the process of discovering bugs in the developed notation, our tool EFinder can be applied to automatically generate relevant examples and to show the Sirius rendering without manual intervention. The underlying idea is that the developer could discover bugs in the editor implementation by observing the models that are rendered automatically and looking for unintended visualizations. The process for generating the constraints for feeding the model finder and for automatically computing the required bounds (i.e., the MV configuration) works by traversing the *odesign model* and by identifying the features that must be present in a model element in order to be shown. For instance, for each *Edge mapping* we need to generate a constraint for each possible pair of source and target types and their associated preconditions. In the example, a Transition could have an initial state or a regular state as source elements and a final state or a regular state as target elements. Thus, we generate the four constraints shown in Fig. 5 (label B).

It is also possible to automatically apply some heuristics to determine proper object bounds. In this case, we know that there must be at least four different instances of Transition, and there might be up to eight constellations of State. Therefore, we automatically advise EFinder with such bounds.

It is worth noting that in this scenario the constraint generation process is driven by the notation to generate OCL IR, which is combined with the AQL to OCL IR translator to process AQL preconditions.

The result, shown in Fig. 5 (label C), is a model which exercises all the elements of the notation. In this example, the generated diagram allows us to discover a bug in the editor specification (or a lack of a meta-model constraint) since it is possible to connect an initial state to a final state directly. The main drawback of the current approach is that, for editors with many notation elements, the generated model may lead to cluttered diagrams. As future work we want to generate classifying terms automatically in order to obtain very small and notation-wise relevant examples.



```
inv InitialState_to_State:
  Transition.allInstances()->exists(o | o.source.isInitial
    and not o.target.isInitial and not o.target.isFinal)

inv State_to_Final:
  Transition.allInstances()->exists(o |
    not o.source.isInitial and not o.source.isFinal and
    o.target.isFinal)

inv Initial_to_Final:
  Transition.allInstances()->exists(o |
    o.source.isInitial and o.target.isFinal)

inv State_to_State:
  Transition.allInstances()->exists(o |
    not o.source.isInitial and not o.source.isFinal and
    not o.target.isInitial and not o.target.isFinal)
```

Figure 5 – Generation of example models from Sirius diagram specifications.

4.3 Cross-artifact analysis

As discussed in previous sections, our architecture is intended to provide model finding support for several OCL variants. However, a useful application of the common OCL IR (intermediate representation) is to carry out an analysis that requires combining different sources, i.e., to enable cross-compatibility between OCL variants. In particular, we have extended the tool AnATLyzer [CGdL17a] to be able to include meta-model invariants as part of its analysis. Let us give an example where both EMF/OCL and ATL OCL are applied: The following listing shows an ATL transformation in which source meta-model invariants are considered via the @constraints annotation. The target meta-model invariant hasKey, written in ATL, can only be verified when combined with the source meta-model invariants, written in EMF/OCL. Technically, the system translates the hasKey invariant defined over the SQL meta-model into a precondition defined over the RelSchema meta-model (using the technique described in [CGdL$^+$17b]). This constraint is fed into the model finder along with the other RelSchema constraints, notably relSchemaKeyNotEmpty, and the result is that the transformation always satisfies the hasKey invariant. To summmarize, we state that our approach is able to *combine* OCL constraints coming from *different* OCL sources into a common representation which is evaluated by the USE MV. Hence, all OCL variants which can be mapped to our IR are automatically compatible with each other. This is, to best of our knowlegde, a unique feature enabled by our framework.

```
-- @path REL=/example.efinder.relschema/models/RelSchema.ecore
-- @path SQL=/example.efinder.relschema/models/SQL.ecore
-- @constraints REL=/example.efinder.relschema/models/RelSchema.ocl
module relschema2sql;
create OUT : SQL from IN : REL;

-- @target_invariant
helper def : hasKey() : Boolean =
  SQL!CreateTable.allInstances()->forAll(ct | ct.pkeys->size() > 0);

```

```
rule RelSchema2Statement {
   from r : REL!RelSchema
     to s : SQL!CreateTable (
    tableName <− '',
    columns <− r.attributes,
    pkeys <− r.attributes→select(att | att.isKey)
  )
}

rule Attribute2ColumnDefinition {
   from a : REL!Attribute
     to c : SQL!ColumnDefinition (
    name <− a.name,
    dt <− a.datetype.name
  )
}
```

Another appealing application of this technique is analyzing model transformation chains written in different languages. For instance, the Eclipse/EMF implementation of QVT Operational makes use of the EMF/OCL variant, ATL implements its own OCL variant, SimpleGT uses SimpleOCL. It would, therefore, be possible to perform an analysis about the composition of a chain of transformations written in different languages by developing an appropriate mapping to our framework. Such a mapping would reuse most of the transformations that we have already developed for existing OCL variants. This is part of our future work.

5 Tool

We have developed an Eclipse plug-in named EFinder which implement the architecture and transformations described in the previous sections. The tool and its source code are available at `http://github.com/jesusc/efinder`.

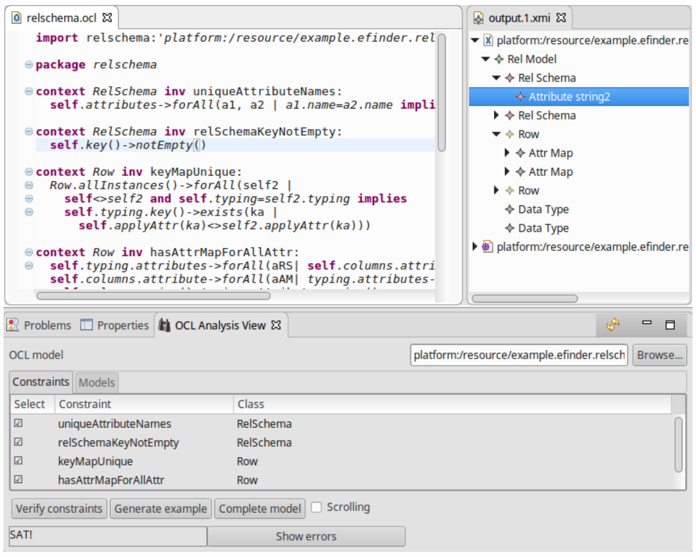
At the level of the *user interface* there is a dedicated OCL View that allows the selection of the available funtionalities: model consistency (plain constraint verification or example generation), partial model completion and scrolling. As a response, one or more EMF models will be generated and stored as XMI files. This is shown in Fig. 6, in which the selected constraints are checked for consistency. The obtained example model demonstrates that there exists at least one model satisfying the constraints. It is shown to the right.

Moreover, EFinder provides an easy to use programmatic interface to allow the developer its integration in external tools using the regular EMF infrastructure. There are three main components involved, whose main APIs are illustrated in the following listing.

1. A translation from the choosen OCL variant to EFinder's OCL IR. We currently provide translations for EMF/OCL, ATL and AQL (lines 1–3).

2. A backend model finder, which is the USE Model Validator in our case. As discussed above, we support partial model completion. Thus it is possible to load a regular EMF Resource and use it as input for the model finder. There are also other options, configuring the scrolling model and whether we want to use a slicing strategy to reduce the search space (lines 5 – 12).

3. The configuration of the model finding process can be configured with transformations to be applied to the IR program and it also support options like setting a time out or the slicing strategy to be applied (lines 13–19).



relschema.ocl

```
import relschema:'platform:/resource/example.efinder.rel

package relschema

context RelSchema inv uniqueAttributeNames:
  self.attributes->forAll(a1, a2 | a1.name=a2.name impli

context RelSchema inv relSchemaKeyNotEmpty:
  self.key()->notEmpty()

context Row inv keyMapUnique:
  Row.allInstances()->forAll(self2 |
    self<>self2 and self.typing=self2.typing implies
    self.typing.key()->exists(ka |
      self.applyAttr(ka)<>self2.applyAttr(ka)))

context Row inv hasAttrMapForAllAttr:
  self.typing.attributes->forAll(aRS| self.columns.attri
  self.columns.attribute->forAll(aAM| typing.attributes-
```

output.1.xmi

```
▼ platform:/resource/example.efinder.re
  ▼ Rel Model
    ▼ Rel Schema
        Attribute string2
    ▶ Rel Schema
    ▼ Row
      ▶ Attr Map
      ▶ Attr Map
    ▶ Row
      Data Type
      Data Type
▶ platform:/resource/example.efinder.re
```

OCL model: platform:/resource/example.efinder.relsch

Select	Constraint	Class
☑	uniqueAttributeNames	RelSchema
☑	relSchemaKeyNotEmpty	RelSchema
☑	keyMapUnique	Row
☑	hasAttrMapForAllAttr	Row

SAT!

Figure 6 – User interface of EFinder

4. Finally, there are several operations to retrieve the result (21–25). First, we can check whether the constraints are satisfiable or not, or whether there is an error and the translation is not possible (line 23). If the result is SAT, the example model can be retrieved as a regular EMF resource (line 24). Moreover, if the scrolling option is active, it is possible to access an iterator to retrieve the next results (line 25).

```
 1 // 1. Loads an EMF/OCL document and translates it to EFinder's IR
 2 ASResource oclSpec = loadOCL("relational.ocl")
 3 EFinderModel ir = EMFOCLTranslator.toIR(oclSpec);
 4
 5 // Load an example model to be used it as partial model
 6 Resource res = rs.getResource(URI.createFileURI("example.xmi"), true);
 7 EMFModel partialModel = new EMFModel(res);
 8
 9 // 2. Creates an instance of USE as a model finder backend
10 IModelFinder finder = new UseMvFinder().
11     withPartialModel(partialModel).
12     withScrolling(ScrollingMode.SCROLL);
13
14 // 3. Configure the model finding process
15 EFinderRunner finder = EFinderRunner.withOCL(ir).
16      withSliceStrategy(SliceMode.NO_SLICE). //Alternative: SLICE_PATH
17      withTransform(new TupleToClassTransform()).
18      withTimeOut(10_000). // 10 seconds
19      withFinder(finder);
20
21 // 4. Get the results
22 Result result = finder.finder();
23 result.getStatus(); // => returns an enumeration SAT/NO-SAT/ERROR
24 result.getWitnessModel(); // => returns EMF Resource with example model
25 result.getScrollingIterator(); // => returns an iterator over EMF Resources
```

6 Evaluation

In this section, we report on the results of the evaluation of this work. In particular, we have focused on measuring to what extent our tool can validate successfully OCL expressions written by third parties. To this end, we have used the data set presented in [NMS17], which is available at `https://github.com/tue-mdse/ocl-dataset`. This dataset contains 509 unique files with constraints. They are Ecore meta-models (.ecore files) or OCL specifications written in EMF/OCL (.ocl files). A total of 301 files are clean files (with no errors), and this valid set contains 2244 OCL constraints. Thus, in average, each OCL model involves around 7 OCL constraints.

The steps followed to process the dataset were as follows. First we read the available metadata descriptors and load the corresponding file. For .ecore files we load the meta-model and extract the embedded OCL expressions using the EMF/OCL API to generate a PivotOCL resource. For .ocl files we load the OCL text using EMF/OCL's CompleteOCL processor and generate a PivotOCL resource as well. In case of load errors or if the OCL text contains syntax errors (e.g., typing errors) we discard the file and count it as "invalid". We also check if the file is duplicated. To do this we compute the MD5 hash of each file and discard duplicates automatically. The remaining files are considered valid from the EMF/OCL point of view, and we continue to process them. The upper part of Table 1 shows the number of valid and invalid files.

Table 1 – Evaluation results

	#Files	%	#Constraints	%
Invalid	208	40.86	0	0,00
Valid	301	59.14	2244	100.00
Total	509	100.00	2244	100.00
Not supported	84	27.91	382	17.02
Internal error	22	7.31	149	6.64
Verified	195	64.78	1713	76.34

Using EFinder we processed each valid file in two ways. First, we tried to verify the consistency of the complete specification (i.e., is it possible to instantiate the meta-model and to satisfy the constraints). We achieved a 64.78% success rate, that is, our tool is able to process a significant part of dataset completely. About 28% of the files contain constructs that are not supported by EFinder and are reported as such. Unfortunately, 22 files resulted in internal errors for which we cannot properly report the cause to the user. Some of these causes require special identification and treatment. For example, we found that USE MV has a stricter type system than EMF/OCL (e.g., USE MV aborts if an oclIsType operation checks an invalid type). Our aim is to avoid these types of issues completely in the future.

Secondly, we processed each constraint individually to avoid the effect that only one instance of an unsupported construct hinders the possibility of evaluating the complete file. In this case the support grows to more than 75%.

To understand better which features are more widely used in the dataset we have automatically processed the valid OCL texts obtaining a summary of used features shown in Table 2. This provides some insights about which characteristics are more important for a model finder to be practical. We have also annotated each feature with its level of support (YES if supported, NO if unsupported and P if partially supported by considering specific cases) in EFinder and EMF2CSP, which is a state-of-the-art tool for the verification of EMF models. This qualitative data comes from a custom test suite in which each construct or operation is exercised with a single, simple expression that test this feature (e.g., to test first test case would include an expression similar to Sequence {1, 2, 3}->first() = 1) For each tool we tested whether it was able to produce a result or if it failed.

As can be observed, String operations are used in a significant number of files (e.g., *size* and *matches*). Another set of unsupported elements has sequence operations, like *at* and *first*. They are generally not supported, neither in EFinder nor EMF2CSP. However, EFinder provides more support for other important features, in particular *oclAsType*, *oclIsUndefined*, *selectByKind*, *collect*, *closure* and *tuples*. This level of support allows EFinder to handle a very relevant set of OCL specifications, as demonstrated in this evaluation.

7 Related work

There are various approaches to validation and verification in the Eclipse and EMF context. The work in [RED18] proposed to validate and verify BPMN processes within Eclipse formally, but did not discuss constraints. [AGGO11] discussed an Eclipse plugin for handling names in conceptual schemas of information systems without treating general constraints. An Eclipse plugin for verifying executable models formulated in ALF for UML was proposed in [PSCG12], however without considering constraints. The approach in [SHU16] demonstrates that constraint solvers (that are the basis for our model finder) can also be used effectively for program and model repair. Graph transformations have been applied in the EMF context for defining, validating and verifying model transformation [BET08, ABJ$^+$10]. Typically these works consider specialized constraint languages and not full standard OCL and its derivations. EMFtoCSP [GBCC12] was the first approach that showed that validation and verification of EMF models, including OCL constraints with solver techniques on the basis of constraint logic programming, is feasible. That approach concentrated on handling the standard OCL features, but put less emphasis on seamless integration into the development process, e.g., for different OCL dialects. An approach to deal with String operations is presented in [BC15]. This was implemented as a CSP problem, but for evaluation purposes it was also tested against Alloy using sequences. The graph solver from [SNV18] based on earlier work on Viatra [BHH$^+$11] also constructs EMF models on the basis of graph or OCL constraints. Complementary to our proposal, the solver is able to generate large, diverse EMF models whereas our aim is to prove model properties with smaller examples or counterexamples. The solver does not support datatype operations (as arithmetic on Integer) and some collection operations (as size). A quite new approach aimed at supporting model finding for practical data generation has been put forward in [SSB19]. Our approach is more closely related to verification, and it is currently limited to generate relatively small models.

There are number of various OCL variants in the EMF ecosystem, such as EMF/OCL [EOT19], AQL [AT19] and ATL [JABK08], SimpleOCL [SOT19] and EOL [KPP06]. All variants share common concepts, but they also have major differences at different levels. EMF/OCL follows the OCL standard, whereas ATL, AQL and SimpleOCL implement it partially or older versions of the OCL standard. EMF/OCL implements a type checker, AQL implements some type checking rules but does not make the typing information available, and ATL, SimpleOCL and EOL just provide runtime interpreters. The type of iterators and operations also varies. ATL does not support operations like selectByKind and closure, which are however available in EMF/OCL for instance. The architecture of EFinder is intended to handle this

Table 2 – OCL features used in the OCL dataset. Features appearing in less than 2% of the files are not shown.

	#Occ.	#Files	%Files	EFinder	EMF2CSP
Operations					
OclAny::oclAsSet	836	87	25,14	YES	YES
OclAny::oclAsType	530	66	19,08	YES	NO
OclAny::oclIsTypeOf	400	60	17,34	YES	YES
OclAny::oclIsUndefined	244	59	17,05	YES	NO
String::size	266	54	15,61	P	YES
OclAny::oclIsKindOf	461	52	15,03	YES	YES
String::matches	143	24	6,94	NO	NO
String::substring	90	24	6,94	NO	NO
String::toUpperCase	24	17	4,91	NO	NO
String::concat	34	13	3,76	NO	NO
OclAny::oclType	28	11	3,18	NO	NO
Integer::toString	23	9	2,60	NO	NO
String::at	15	8	2,31	NO	NO
String::toLowerCase	10	8	2,31	NO	NO
Collection ops.					
size	732	149	43,06	YES	YES
includes	363	89	25,72	YES	YES
isEmpty	335	72	20,81	YES	YES
notEmpty	229	68	19,65	YES	YES
asSet	182	32	9,25	YES	NO
excludes	66	31	8,96	YES	YES
includesAll	87	30	8,67	YES	NO
union	116	24	6,94	YES	YES
sum	64	20	5,78	YES	NO
first	72	18	5,20	NO	NO
at	141	17	4,91	NO	NO
excluding	35	15	4,34	YES	P
prepend	15	15	4,34	NO	NO
selectByKind	84	14	4,05	YES	NO
including	53	13	3,76	YES	YES
indexOf	35	12	3,47	NO	NO
intersection	27	11	3,18	YES	YES
asSequence	37	10	2,89	P	YES
excludesAll	27	10	2,89	YES	YES
flatten	49	10	2,89	YES	YES
last	15	10	2,89	NO	NO
selectByType	24	9	2,60	YES	NO
asOrderedSet	35	8	2,31	NO	YES
symmetricDifference	8	7	2,02	YES	NO
Iterators					
forAll	993	154	44,51	YES	YES
select	1085	138	39,88	YES	YES
collect	2210	106	30,64	YES	NO
exists	793	73	21,10	YES	YES
isUnique	180	62	17,92	P	YES
one	580	27	7,80	YES	YES
closure	99	25	7,23	YES	NO
any	99	24	6,94	YES	YES
reject	9	7	2,02	YES	YES
Iterate	22	13	3,76	YES	NO
Tuples	180	39	11,27	YES	NO
Ranges	62	24	6,94	P	NO

variability in order to widen the application scope of the tool.

In [AZKR17] an initial version of our infrastructure was used to build a symbolic executor for ETL. This shows the usefulness of our framework for researchers to build advanced tooling with a fraction of the effort required to build it from scratch. In contrast to the mentioned works, our current proposal is the only approach for EMF models that supports advanced model finding techniques with various analysis scenarios and is open for different OCL versions.

8 Conclusions

We have presented EFinder, a framework to provide model finding capabilities for the EMF ecosystem. We have applied the USE MV as a model finder backend, enhancing its capabilities through dedicated transformations. Our architecture, which is based on an intermediate OCL representation, provides support for several OCL variants. We have shown its usefulness in three non-trivial scenarios. Finally, the evaluation results point out that EFinder is able to process about 65% of the OCL files present in a third-party dataset, covering a wide range of OCL features used in practice. This indicates that EFinder is already a practical tool.

As future work, we plan to enhance the integration of EFinder with the USE MV. We want to support some scenarios supported by USE MV but not supported by EFinder, notably Partitioning with Classifying Terms. At the USE MV side, we aim at exploring to what extent it is possible to implement widely used OCL operations (according to the results in Table 2) which are not supported yet, notably String operations. Last but not least, larger practical case studies have to give feedback for further improvement of our model finding option in the EMF context.

Acknowledgments Work partially funded by project RECOM (Spanish MINECO, TIN2015-73968-JIN, AEI/FEDER/UE) and a Ramón y Cajal 2017 research grant.

References

[ABJ$^+$10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems - 13th Int. Conf., Proceedings, Part I*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010. doi:10.1007/978-3-642-16145-2_9.

[AGGO11] David Aguilera, Raúl García-Ranea, Cristina Gómez, and Antoni Olivé. An eclipse plugin for validating names in UML conceptual schemas. In Olga De Troyer, Claudia Bauzer Medeiros, Roland Billen, Pierre Hallot, Alkis Simitsis, and Hans Van Mingroot, editors, *Advances in Conceptual Modeling. Recent Developments and New Directions - ER 2011 Workshops FP-UML, MoRE-BI, Onto-CoM, SeCoGIS, Variability@ER, WISM. Proceedings*, volume 6999 of *LNCS*, pages 323–327. Springer, 2011. doi:10.1007/978-3-642-24574-9_41.

[AT19] AQL-Team. AQL: Acceleo Query Language. https://www.eclipse.org/acceleo/documentation/aql.html/, 2019. [Online; accessed April-2019].

[AZKR17] Banafsheh Azizi, Bahman Zamani, and Shekoufeh Kolahdouz-Rahimi. Contract verification of etl transformations. In *2017 7th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 154–160. IEEE, 2017.

[BC15] Fabian Büttner and Jordi Cabot. Lightweight string reasoning in model finding. *Software & Systems Modeling*, 14(1):413–427, 2015.

[BET08] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise semantics of EMF model transformations by graph transformation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th Int. Conf., MoDELS 2008. Proceedings*, volume 5301 of *LNCS*, pages 53–67. Springer, 2008. `doi:10.1007/978-3-540-87875-9\_4`.

[BHH+11] Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Implementing efficient model validation in EMF tools. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, pages 580–583. IEEE Computer Society, 2011. `doi:10.1109/ASE.2011.6100130`.

[CGdL17a] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Static analysis of model transformations. *IEEE Transactions on Software Engineering*, 43(9):868–897, 2017.

[CGdL+17b] Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara, Robert Clarisó, and Jordi Cabot. Translating target to source constraints in model-to-model transformations. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 12–22. IEEE, 2017.

[Ecl17] EclipseTeam. Eclipse Modeling Framework (EMF) Model Validation. www.eclipse.org/emf-validation/, 2017.

[EOT19] EMF-OCL-Team. Eclipse OCL (Object Constraint Language). `https://download.eclipse.org/ocl/doc/6.4.0/ocl.pdf`, 2019. [Online; accessed April-2019].

[GBCC12] Carlos A González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. Emftocsp: A tool for the lightweight verification of emf models. In *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*, pages 44–50. IEEE, 2012.

[GHD18] Martin Gogolla, Frank Hilken, and Khanh-Hoang Doan. Achieving model quality through model validation, verification and exploration. *Computer Languages, Systems & Structures*, 54:474–511, 2018.

[HGBV18] Frank Hilken, Martin Gogolla, Loli Burgueño, and Antonio Vallecillo. Testing models and model transformations using classifying terms. *Software & Systems Modeling*, 17(3):885–912, 2018.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[KG12] Mirco Kuhlmann and Martin Gogolla. From uml and ocl to relational logic and back. In *International Conference on Model Driven Engineering Languages and Systems*, pages 415–431. Springer, 2012.

[KPP06] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The Epsilon Object Language (EOL). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006.

[NMS17] Jeroen Noten, Josh GM Mengerink, and Alexander Serebrenik. A data set of ocl expressions on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 531–534. IEEE, 2017.

[PSCG12] Elena Planas, David Sanchez-Mendoza, Jordi Cabot, and Cristina Gómez. Alf-verifier: An eclipse plugin for verifying alf/uml executable models. In Silvana Castano, Panos Vassiliadis, Laks V. S. Lakshmanan, and Mong-Li Lee, editors, *Advances in Conceptual Modeling - ER 2012 Workshops CMS, ECDM-NoCoDA, MoDIC, MORE-BI, RIGiM, SeCoGIS, WISM. Proceedings*, volume 7518 of *LNCS*, pages 378–382. Springer, 2012. `doi:10.1007/978-3-642-33999-8\_44`.

[RED18] Anass Rachdi, Abdeslam En-Nouaary, and Mohamed Dahchour. Short paper: BPMN process analysis: A formal validation and verification eclipse plugin for BPMN process models. In Andreas Podelski and François Taïani, editors, *Networked Systems - 6th Int. Conf., NETYS 2018, Revised Selected Papers*, volume 11028 of *LNCS*, pages 100–104. Springer, 2018. `doi:10.1007/978-3-030-05529-5\_7`.

[SHU16] Friedrich Steimann, Jörg Hagemann, and Bastian Ulke. Computing repair alternatives for malformed programs using constraint attribute grammars. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016*, pages 711–730. ACM, 2016. `doi:10.1145/2983990.2984007`.

[SNV18] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. A graph solver for the automated generation of consistent domain-specific models. In *Proceedings of the 40th International Conference on Software Engineering*, pages 969–980. ACM, 2018.

[SOT19] Simple-OCL-Team. SimpleOCL. `https://github.com/dwagelaar/simpleocl`, 2019. [Online; accessed April-2019].

[SSB19] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C Briand. Practical model-driven data generation for system testing. *arXiv preprint arXiv:1902.00397*, 2019.

About the authors

Jesús Sánchez Cuadrado is a Ramón y Cajal researcher at Universidad de Murcia. Previously he was an associate professor at Universidad Autónoma de Madrid. His research is focused on Model Driven Engineering (MDE) topics, notably model transformation languages, meta-modelling and domain specific languages. On these topics, he has published several articles in journals and peer-reviewed conferences, and developed several tools. Contact him at `jesusc@um.es`, or visit `http://sanchezcuadrado.es`.

Martin Gogolla is professor for Computer Science at University of Bremen, Germany and is the head of the Research Group Database Systems. His research interests include software development with object-oriented approaches, formal methods in system design, semantics of languages, and formal specification. Martin Gogolla is actively participating in the MODELS community and is involved in the organisation of the OCL workshops. Martin Gogolla is Associate Editor of the Springer journal on Software and Systems Modeling. In his group, foundational work on the semantics of and the tooling for UML, OCL and general modeling languages has been carried out. The group develops the OCL and UML tool USE (UML-based Specification Environment) since about 15 years. The tool is internationally and nationally widely accepted and employed for research and teaching and in software production. Contact him at `gogolla@informatik.uni-bremen.de`