Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 152 (2006) 69–81
www.elsevier.com/locate/entcs

Using ATL for Checking Models[1]

Jean Bézivin Frédéric Jouault

ATLAS Group, INRIA and LINA,
University of Nantes, France
{jean.bezivin,frederic.jouault}@univ-nantes.fr

Abstract

Working with models often requires the ability to assert the compliance of a given model to a given set of constraints. Some tools are able to check OCL invariants on UML models. However, there are very few tools able to do the same for any metamodel. This is quite penalizing for the DSL (Domain Specific Language) approach to model engineering. In this paper we propose a metamodel-independent solution to this problem that uses ATL (ATLAS Transformation Language). This solution has been implemented as an Eclipse-based plugin.

Keywords: Model Engineering, OCL, checking models, ATL

1 Introduction

Model engineering is based on the notion of models conforming to explicit metamodels [4]. A metamodel constrains the structure of a model: elements of a model are typed by elements of its metamodel. However, this is often not enough in practice. Additional constraints need to be specified. For instance, if a metamodel simply defines the `age` attribute of a person as an integer, invalid (e.g. negative) values could be used. In response to this requirement, the OMG (Object Management Group) specified OCL [9] (Object Constraint Language) to express constraints on models.

Currently, UML [11] is the most widely used metamodel and is supported by a large number of tools. UML is aimed at modelling software artefacts and can be adapted to specific domains through the use of profiles. OCL support

[1] Work partially supported by ModelWare, IST European project 511731.

doi:10.1016/j.entcs.2006.01.015

varies from one UML tool to another. The most advanced are able to evaluate (or compile) constraints.

But model engineering is not only about UML. The DSL [6] (Domain Specific Language) approach promotes the definition of a large number of small domain-specific metamodels, rather than using a single and large metamodel. At the present time, few DSL tools are able to evaluate constraints on models. The objective of this work is to show how existing model transformation tools, such as ATL [2] [1], can be used for this purpose.

In [4] we argued that the principle "everything is a model" may be viewed as the central driving force in the present evolution of model engineering. In [2] we pursued this by demonstrating how concretely a transformation between models may be itself regarded as a model. Here we try to go one step further by showing that a verification itself can be considered likewise.

Traditionally a verification is a function returning a *Boolean* from a *Model*. If we call f the verification function, this can be noted $f : Model \rightarrow Boolean$. Constraints can be extracted from f in the form of a specific type of *Model*: *Constraints*, to get $f' : Model \times Constraints \rightarrow Boolean$. To indicate an error severity level, the result of the verification can be extended to an *Integer* as in $f'' : Model \times Constraints \rightarrow Integer$. We propose here another extension, which is to consider the result as a specific kind of *Model*: a *Diagnostic*, to obtain $f''' : Model \times Constraints \rightarrow Diagnostic$.

The structure of the paper is the following: section 2 presents a sample DSL used as an illustration. In section 3, we analyze some requirements for well-formedness rules on models. Section 4 explores the use of models to represent the result of model verification. Section 5 shows how ATL can be used to check constraints on models. Section 6 presents related work.

2 Motivating Example: Class Diagrams

Throughout this paper we will use a very simple DSL called CD for Class Diagrams. This has been chosen for reason of brevity, but more complex DSLs could be used similarly. A metamodel of CD, is given in Fig. 1. CD can be considered as a simplified subset of UML. Every element of a class diagram has a name. *Classes* can have supertypes and *StructuralFeatures*, which are *References* to other *Classes* or *Attributes*. *StructuralFeatures* have a multiplicity and are typed by a *Class* or a *DataType*. *Packages* are used to structure diagrams by grouping related elements.

The definition given in Fig. 1 is clearly structural and still incomplete. It is thus possible to create models conforming to CD that are however not valid class diagrams. The specification of CD can be improved with the addition

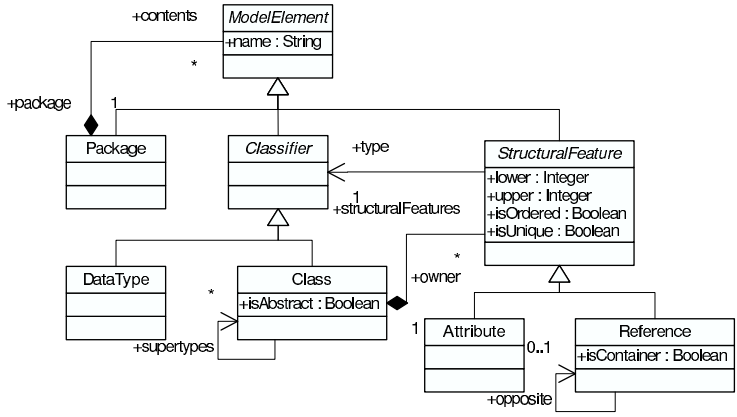

Fig. 1. A simple class diagram metamodel

of two constraints (C1) and (C2): *Classifier* names must be unique within a *Package* (C1) and *StructuralFeature* names must be unique within a *Class* and its supertypes (C2).

A complete definition of CD would take some place and is out of the scope of this paper. We are rather going to limit ourselves to a small number of constraints, including (C1) and (C2). The focus is on giving usable concrete expressions of them.

3 Well-formedness Rules for Model Engineering

In order to be automatically verified, constraints must be written in a language for which automatic translation to an executable form is possible. The well-known OCL solution is used here. In this section, we extend simple OCL constraints by adding severities and descriptions. The result can be used to specify well-formedness rules on models.

3.1 Using OCL to Express Constraints

OCL basically knows three kinds of constraints. Invariants must be verified at all times a model is supposed to be in a consistent state. Pre- and post-conditions are specified on operations and must respectively be valid before and after the execution of their bodies. Only some metamodels have execution semantics and therefore need such constraints. Since this is not the general case, we will only consider constraints of the first kind.

An invariant is defined in the context of a metamodel type. It is composed of a boolean expression, which must evaluate to true for every element of this type. Fig. 2 gives OCL specifications of (C1) and (C2). The `allStructuralFeatures()` operation on Class returns the collection of all

```
-- (C1) Error: the name of a Classifier must
-- be unique within its package.
context Classifier inv:
  not self.package.contents->exists(e |
    (e <> self) and (e.name = self.name))

-- (C2) Error: the name of a StructuralFeature must
-- be unique within its Class and its supertypes.
context StructuralFeature inv:
  not self.owner.allStructuralFeatures()->exists(e |
    (e <> self) and (e.name = self.name))
```

Fig. 2. Expression of (C1) and (C2) as OCL invariants

the *StructuralFeatures* belonging to the class and to all its supertypes. It can be defined in OCL too, as listed in Appendix (section 8).

3.2 Adding Severities to Constraints

The two invariants (C1) and (C2) *must* be verified for every CD model. It is however sometimes useful to specify constraints that *should* not be violated. When such constraints are not verified, the consistency of the model is not provably wrong. This is quite similar to the way compilers traditionally tag messages as *error* or *warning*. An error is fatal while a warning indicates a potential problem. More degrees can be defined in this scale as we will see in the following paragraph. We will now speak of this as the *severity* of a constraint. According to this definition, (C1) and (C2) are errors, because a model that violates them is not a valid class diagram.

Let us consider the two following constraints: in class diagrams, an abstract class *should* have children (C3) and the name of a Classifier *should* begin with an upper case letter (C4). A CD model which violates (C3) or (C4) is still a class diagram. These constraints are therefore not errors. The precise severity associated to a non-error constraint can vary. In this example, we consider (C3) to be a *warning*, as unreachable code is in some programming languages. (C4) is called a *critic* here since it is not an error and does not impact the structure of the metamodel at all. It is merely a style consideration. Fig. 3 gives OCL expressions for these two constraints.

3.3 Adding Descriptions to Constraints

So far, the violation of an invariant can only be associated to the constraint itself, its severity and the violating model element. It is however not always

```
-- (C3) Warning: an abstract class should have children.
context Class inv:
  not (self.isAbstract and
      (Class.allInstances()->select(e |
        e.supertypes->includes(self)
      )->size() = 0))

-- (C4) Critic: the name of a Classifier should
-- begin with an upper case letter.
context Classifier inv:
  not (let firstChar : String =
      self.name.substring(1, 1) in
  firstChar <> firstChar.toUpper())
```

Fig. 3. Expression of (C3) and (C4) as OCL invariants

```
'The Class ' + self.owner.name + ' contains several ' +
'properties having the same name: ' + self.name + '.'
```

Fig. 4. An OCL expression evaluating to a context-adapted error message when associated to (C2)

easy to understand the issue with only the boolean expression associated to the constraint. A description should consequently be associated to each constraint. This has been done for (C1) and (C2) in Fig. 2 and for (C3) and (C4) in Fig. 3 in the form of comments. It is however not enough, since a tool cannot use such freeform specification.

In UML, constraints have a name that can be used to roughly indicate the semantics of the constraint as a human readable string. This is a first step. But in some cases, the text of the message may need to be adapted to the context. For instance, it could contain references to the value of some properties of violating elements (e.g. their names). The solution we use in this paper is to specify the description as an OCL expression evaluating to a string. An example is given in Fig. 4 for (C2). However, since a diagnostic is a model, it is possible to generate a more precise and structured representation of the issue if this is necessary.

3.4 Extending OCL

As we previously observed, tools need to access the severities and descriptions of constraints. Otherwise, specifications like those given in Fig. 3 cannot be distinguished from those given in Fig. 2. OCL therefore needs extensions to support additional elements attached to constraints.

Fig. 5 gives an extended version of (C1) specifying, in addition to its

```
context CD!Classifier report error
    'a Classifier of the same name already ' +
    'exists in the same package: ' + self.name
  at self.getLocation()
  if self.package.contents->exists(e |
     (e <> self) and (e.name = self.name));
```

Fig. 5. Specification of (C1) with OCL extensions

context and its boolean expression:

- A severity, indicating the failing degree of the problem. It is here an *error*, but could also be a *warning* or a *critic* for instance.
- A description, stating the problem in a human understandable way. Note the use of a dynamically built string using the name of the violating element.
- A string encoding the location of the problem in a computer-readable format. The choice of the format to use is tool-dependent and can be implemented in a `getLocation()` operation.

4 Representing a Verification Result as a Model

Verifying a set of constraints on a model results in a *diagnostic*. In this section, we consider several representations of diagnostics and give an example metamodel.

4.1 Different Representations of Diagnostics

The simplest form of diagnostic is a boolean. The `true` value means the model satisfies all the constraints whereas `false` means the model fails to statisfy all of them. If the diagnostic is represented as an integer, it can be used to encode the failure degree. Its value could, for instance, be the number of constraint violations, optionally weighted by the severity of each constraint. This is a bit more interesting, but not very helpful to point out the natures and locations of problems.

More complex representations are desirable. Some tools present the diagnostic to the user textually (e.g. compilers), embedded in a GUI (Graphical User Interface), etc. We suggest to represent the diagnostic as a model. The structure of the diagnostic is therefore explicitly specified as a metamodel.

Since the diagnostic is a model, any transformation can be performed on it, including extraction to a text or XML representation. For instance, using the template: "<location>: <severity>: <description>", we get a notation similar to the one used used by many compilers. Within an IDE, the diagnostic

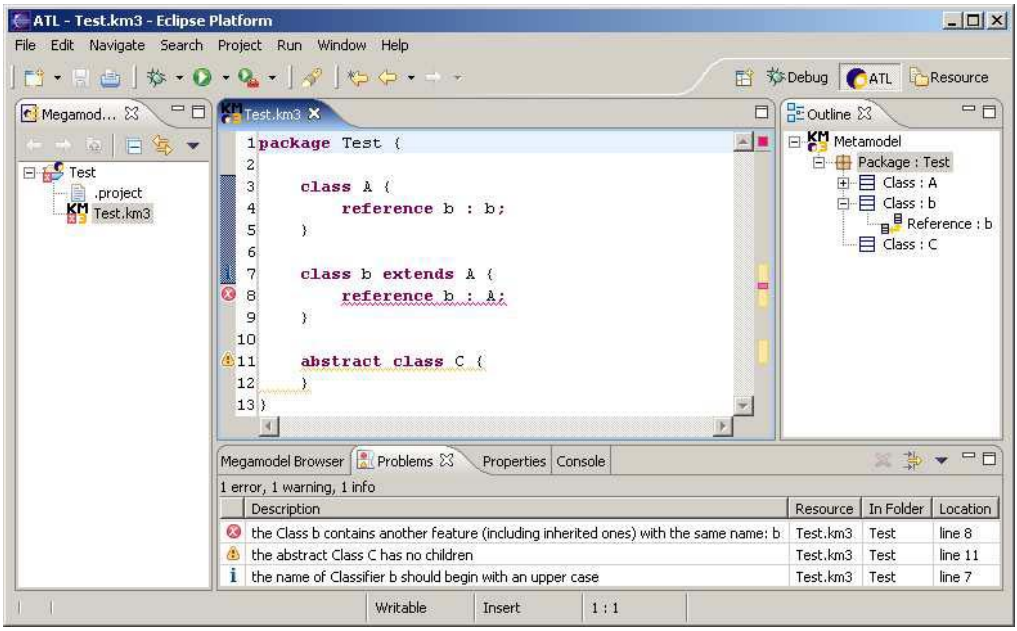

Fig. 6. A screenshot of the Eclipse prototype

model can be mapped to the native representation (e.g. IMarkers in Eclipse). The problems then show up at the corresponding location in the editor and in the "Problems" view, as shown on Fig. 6. Many other representations of the diagnostic may also be derived.

4.2 A Metamodel for Diagnostics

An element can either pass or fail the verification of a constraint. The first case is considered normal and need not be memorized. In the second case, a problem has been identified. The metamodel of Fig. 7 specifies a *Problem* class to represent such a problem.

A model satisfies a whole set of constraints when their evaluation produces an empty Problem model. Each *Problem* bears the severity, typed using an enumeration, associated to the constraint. It also contains a description and a location, represented by strings. The number of severity levels and the representation of the location and description of the problem may be tuned to fit specific needs.

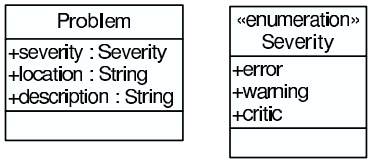

Fig. 7. A sample metamodel for diagnostics: Problem

5 Using the ATL Engine to Verify Constraints

5.1 Presentation of ATL

A model transformation is an automatic generation of a set of target models from a set of source models. The upcoming OMG standard for model transformation is called QVT [10] (Query/View/Transformation). ATL is a QVT-like model transformation language. An execution engine for ATL is available as an open-source (EPL license) Eclipse set of plugins on the GMT website [7]: execution engine, transformation editor, source-level debugger, etc.

ATL is a hybrid of declarative and imperative. Only the declarative part is of interest to this paper. The following definitions describe only a subset of ATL (i.e. declarative only, single source element by rule, etc.) necessary to understand what comes next.

An ATL transformation program consists of transformation rules. Each rule contains a source pattern to be matched in the source models and a target pattern to be created in the target models for each match. The source pattern has a type coming from a source metamodel and a guard, in the form of an OCL boolean expression. A match of a rule corresponds to an element of a source model that is typed by the source pattern type of the rule and for which the guard evaluates to `true`. The target pattern is composed of a set of types coming from the target metamodels. Every type is associated to a set of bindings, each of them specifying how to initialize one of its properties from an OCL expression. For each match of a rule, one element for each target pattern type is created in the target models. Each target element is then initialized by applying the corresponding bindings.

5.2 ATL Representation of Constraints

The ATL engine can only be used to verify constraints if they are expressed as was just specified. The algorithm to create a checking program from a set of constraints is the following: for each constraint, create an ATL transformation rule so that:

- The source pattern type of the rule is the context of the constraint.
- The guard of the rule contains the negation of the boolean expression as-

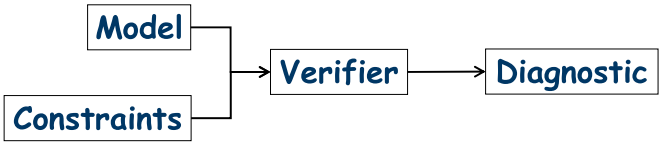

Fig. 8. The Verifier

sociated to the constraint. Note that we expressed (C1) and (C2) in Fig. 2 and (C3) and (C4) in Fig. 3 as negations (i.e. of the form `not <expr>`). We indeed anticipated that double negations would disappear, resulting in a simpler program.

- The target pattern of the rule specifies a single type: *Problem*, that is to be created on a match (i.e. on a violation of the invariant).
- It will be initialized using three bindings for: the severity of the invariant, a description of the issue and its location. Since the target element is initialized from OCL expressions navigating the source model, the implementation of a description as a constructed string is straightforward.

The source of an ATL transformation implementing this solution for (C1), (C2), (C3) and (C4) is given in Appendix (section 8). We implemented the verification of a slightly larger set of constraints on CD models in an Eclipse plugin using the Eclipse Modeling Framework [5]. We chose to represent class diagrams textually, using a simple syntax. In this case, the `location` of a *Problem* is therefore composed of a line and a column number. Fig. 6 shows what is actually presented to the user when a class diagram contains problems.

5.3 Analysis of the Solution

Fig. 8 presents a generic structure for constraints verification. A *Verifier* checks *Constraints* on a *Model*, which results in a *Diagnostic*. Let us note V the operation implemented by the *Verifier*, M the *Model* to verify, C the *Constraints* and D the resulting *Diagnostic*. We can then write $D = V(M, C)$.

We have just presented a solution integrating the constraints and the verifier in the same entity: an ATL program. If we note P this program, then we have $D = P(M)$. By comparing the two expressions of D we observe that P corresponds to V in which C is already evaluated. This is very similar to the currying operation of functional languages. As a matter of fact, a curried version of V would be noted $V_C(M)$ such as $D = V_C(M)$.

A given set of constraints C is very likely to be used to check several models conforming to the corresponding metamodel. It makes sense to not reevaluate all constraints each time. This is what we do in practice when we use the ATL compiler to compile the constraints.

6 Related Work

The work presented in this paper is related to the notion of *code smells* [8]. A smell indicates a problematic fragment in code. This notion can be adapted to models. Thus, in [12], the authors show how OCL expressions can be used to specify such smells on UML models.

With the solution presented in this paper, evaluation of code smells on different kind of models is possible. The detection of smells can be implemented in OCL, using all the power of this language (iterators, recursion, etc.). The target metamodel of the smell detector can be adapted to fit existing tools. The ATL engine can be used to detect the specified code smells.

The concept of smell is often associated to the notion of refactoring. When smells are detected on a model, appropriate actions can be taken in order to get rid of the problem. Such refactorings could also be implemented as model transformation. The work presented here shows thus how a complete automated refactoring chain could be implemented.

7 Conclusions and Future Work

We have made three suggestions to improve constraints checking in model engineering:

- Constraints, which have a context and a boolean expression in OCL, are associated to additional information. This can be, for instance, a severity, a description, a location, etc.
- The diagnostic resulting from the verification of a set of constraints on a model is considered as a model. This model can then be transformed into any representation: textual, graphical, embedded in a GUI, etc.
- The ATL language can be used to express constraints on models. The ATL compiler is then used to compile them and the ATL engine to perform the verification.

This approach can be used for any metamodel and in several contexts. For instance, to verify the validity of a UML model with respect to a given profile. Another possible use is to protect the execution of a model transformation. As a matter of fact, transformation programs often make some assumptions about their source models that are not part of their metamodels. By first executing a checking program, users may be warned of incorrect uses of transformations.

The solution can also be adapted to the computation of metrics on models. The target metamodel needs to be extended to support the required representation of the results. OCL expressions can be used to specify how

to compute the metrics. These expressions are then embedded into an ATL transformation.

In all these cases, complex nets of relations appear between metamodels, UML profiles, generic transformations, constraint checking transformations, etc. There is clearly a need to deal with large numbers of such entities. The notion of megamodel [3] is a possible solution to handle this additional complexity.

References

[1] ATLAS group, ATLAS Transformation Language, Reference site: http://www.sciences.univ-nantes.fr/lina/atl/.

[2] Bézivin, J., Dupé, G., Jouault, F., Pitette, G., and Rougui, J. E., First experiments with the ATL model transformation language: Transforming XSLT into XQuery, 2nd OOPSLA Workshop on Generative Techniques in the context of MDA, Anaheim, CA, USA (2003).

[3] Bézivin, J., Jouault, F., Valduriez, P., On the Need for Megamodels, OOPSLA Workshop on Best Practices for Model-Driven Software Development, Vancouver, BC, Canada (2004).

[4] Bézivin, J., *On the unification power of models*, Software and Systems Modeling, Volume **4**, Issue 2, May 2005, pages 171 - 188.

[5] Budinsky, F., Steinberg, D., Raymond Ellersick, R., Ed Merks, E., Brodsky, S. A., Grose, T. J., "Eclipse Modeling Framework", Addison Wesley (2003).

[6] Czarnecki, K., Eisenecker, U., "Generative Programming: Methods, Tools, and Applications", Addison-Wesley (2000).

[7] Eclipse Foundation, "Generative Model Transformer project", Reference site: http://www.eclipse.org/gmt/.

[8] Fowler, M., "Refactoring - Improving the Design of Existing Code", Addison Wesley, July 1999.

[9] OMG, "UML OCL 2.0 Specification", OMG Adopted Specification ptc/03-10-14 (2003), http://www.omg.org/docs/ptc/03-10-14.pdf.

[10] OMG, Request For Proposal: "MOF 2.0 Query / Views / Transformations RFP", ad/2002-04-10 (2002), http://www.omg.org/cgi-bin/doc?ad/2002-4-10.

[11] OMG, "UML 2.0 Superstructure Specification", OMG Adopted Specification ptc/03-08-02 (2003), http://www.omg.org/docs/ptc/03-08-02.pdf.

[12] Van Gorp, P., Stenten, H., Mens, T., Demeyer, S., Towards automating source-consistent UML refactorings, In Proceedings of UML 2003 Conference, Springer-Verlag LNCS 2863, San Francisco, CA, October 2003, pages 144 - 158.

8 Appendix

```
-- ATL transformation that verifies (C1), (C2),
-- (C3) and (C4) on CD (i.e. Class Diagram) models.
module CD_Verifier;
```

```
create OUT : Problem from IN : CD;

-- (C1) Error: the name of a Classifier must
-- be unique within its package.
rule ClassifierNameUniqueInPackage {
  from
    i : CD!Classifier (
      i.package.contents->exists(e |
       (e <> i) and (e.name = i.name)))
  to
    o : Problem!Problem (
      location <- i.location, severity <- Severity::error,
      description <- 'a Classifier of the same name ' +
          'already exists in the same package: ' + i.name
    )
}

helper context CD!Class def:
  allStructuralFeatures() : Sequence(CD!StructuralFeature) =
        self.structuralFeatures->union(
          self.supertypes->collect(e |
            e.allStructuralFeatures())->flatten());

-- (C2) Error: the name of a StructuralFeature must
-- be unique within its Class and its supertypes.
rule StructuralFeatureNameUniqueInClass {
  from
    i : CD!StructuralFeature (
      let sfs : Sequence(CD!StructuralFeature) =
          i.owner.allStructuralFeatures() in
      sfs->exists(e | (e <> i) and (e.name = i.name)))
  to
    o : Problem!Problem (
      location <- i.location, severity <- Severity::error,
      description <- 'the Class ' + i.owner.name +
          ' contains another feature (including ' +
          'inherited ones) with the same name: ' + i.name
    )
}

-- (C3) Warning: an abstract class should have children.
```

```
rule AbstractClassShouldHaveChildren {
  from
    i : CD!Class (
      i.isAbstract and
      (CD!Class.allInstances()->select(e |
        e.supertypes->includes(i))->size() = 0))
  to
    o : Problem!Problem (
      location <- i.location, severity <- Severity::warning,
      description <- 'the abstract Class ' + i.name +
          ' has no children'
    )
}

-- (C4) Critic: the name of a Classifier should
-- begin with an upper case letter.
rule ClassifierNameShouldStartWithUpperCase {
  from
    i : CD!Classifier (
      let firstChar : String = i.name.substring(1, 1) in
      firstChar <> firstChar.toUpper())
  to
    o : Problem!Problem (
      location <- i.location, severity <- Severity::critic,
      description <- 'the name of Classifier ' + i.name +
          ' should begin with an upper case'
    )
}
```