

# Operation Based Model Representation: Experiences on Inconsistency Detection

Jerome Le Noir<sup>1</sup>, Olivier Delande<sup>1</sup>, Daniel Exertier<sup>2</sup>, Marcos Aurélio Almeida da Silva<sup>3</sup>, and Xavier Blanc<sup>4</sup>

<sup>1</sup> Thales Research and Technology, France

<sup>2</sup> Thales Corporate Services, France

<sup>3</sup> LIP6, UPMC Paris Universit as, France

<sup>4</sup> LABRI, Universit e de Bordeaux 1, France

**Abstract.** Keeping the consistency between design models is paramount in complex contexts. It turns out that the underlying Model Representation Strategy has an impact on the inconsistency detection activity. The Operation Based strategy represents models as the sequence of atomic editing actions that lead to its current state. Claims have been made about gains in time and space complexity and in versatility by using this kind of representation when compared to the traditional object based one. However, this hypothesis has never been tested in an industrial context before. In this paper, we detail our experience evaluating an Operation Based consistency engine (Praxis) when compared with a legacy system based on EMF. We evaluated a set of industrial models under inconsistency rules written in both Java (for EMF) and PraxisRules (the DSL – Domain Specific Language – for describing inconsistency rules in Praxis). Our results partially confirm the gains claimed by the Operation Based engines.

## 1 Introduction

Current large scale software projects involve hundreds of developers working in a distributed environment over several models that need to conform to several meta-models (e.g. SysML, UML, Petri nets, business process) [1]. In such context keeping the consistency between models and with their respective meta-models is mandatory[2].

Models are usually represented as sets of *objects* along with their *attributes* and mutual *associations* [3, 4]. A model is considered to be inconsistent if and only if it contains undesirable patterns, which are specified by the so called *inconsistency rules* [5]. Even if these rules may be represented in many different ways, such as the well-formedness rules of [6], the structural rules of [7], the detection rules of [4], the syntactic rules of [8], and the inconsistency detection rules of [9], approaches that deal with detection of inconsistencies irremediably consist in browsing the model in order to detect undesirable patterns.

The underlying strategy used to represent the model is then very likely to have significant effects on the performance of inconsistency detection algorithms.

Under the Operation Based model representation strategy[9, 10], instead of keeping track of the current configuration of the objects, their attributes and associations, one records the sequence of atomic editing actions that were performed in order to obtain the current configuration.

Operation Based checkers claim to be as efficient as Object Based ones, and very adequate for Incremental inconsistency detection mode[11]. This mode, consists in, instead of checking every inconsistency rule over the complete model every time the model has been modified, limiting the search to a subset of this problem. The search for inconsistencies is performed on the subset of the model that was modified since the last check and using the subset of the inconsistency rules that are concerned by this modification. The efficiency gains claimed by these checkers would come from the fact that identifying the scope of an incremental check reduces to looking at the sequences of actions appended to the current model by the uses (these sequences are called *increments*)[12].

Unfortunately, none of these claims has ever been tested in an industrial context before, specially on non-UML models. In this paper we report our experiences on the impact of the underlying strategy for model representation to the overall performance of the inconsistency checker. These experiences have been synthesized in a case study in which we compare an operation based consistency checker (Praxis [9]) with the one provided by the Eclipse Modeling Framework (EMF)<sup>5</sup>. Our tests included industrial models ranging from 1.000 to 50.000 model elements. We have carried out the approach on the engineering meta-model defined by Thales composed of about 400 meta-classes. This meta-model contains 114 inconsistency rules implemented in Java, from which 30 mandatory ones were selected, re-implemented and checked in models coming from operational contexts. Our results partially confirm the gains claimed by Operation Based consistency engines: the overall performance gains were identified in the incremental mode but no significant gains were identified in the batch mode.

This paper is organized as follows: Section 2 details the operation model representation strategy used in Praxis and compares it to the object based one used in EMF. Section 3 presents the design and results of our case study. Section 4 concludes.

## 2 Praxis: An Operation Based Model Representation Strategy

The objective of this section is introducing Praxis, an operation based model representation strategy and PraxisRules, the rule based DSL – Domain Specific Language – for representing consistency rules in Praxis. Our objective is not presenting them in details, but to present their basics in comparison to traditional object based model representation and consistency rules as provided by EMF, for example.

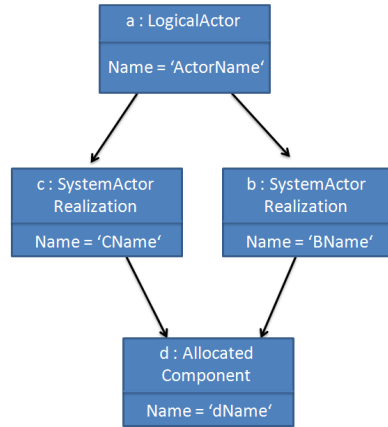
---

<sup>5</sup> The Eclipse Modeling Framework, <http://eclipse.org/emf>

## 2.1 Praxis

Praxis[9] is a meta-model independent consistency checking strategy whose internal model representation is based on the operation based model representation. In Praxis, a model is represented as the sequence of atomic editing actions that lead to its current state. This approach uses 6 kinds of atomic actions which were inspired on the MOF reflective API [3].

The  $create(me, mc, t)$  and  $delete(me, t)$  actions respectively create and delete a model element  $me$ , that is an instance of the meta-class  $mc$  at the time-stamp  $t$ . The time-stamp which indicates the moment when it was executed by the user. The actions  $addProperty(me, p, value, t)$  and  $remProperty(me, p, value, t)$  add or remove the value  $value$  to or from the property  $p$  of the model element  $me$  at time-stamp  $t$ . Similarly, the actions  $addReference(me, r, target, t)$  and  $remReference(me, r, target, t)$  add or remove the model element  $target$  as a reference  $r$  from the model element  $me$ .



**Fig. 1.** Sample model

```

create(a,logicalActor,1)
addProperty(a,name, 'ActorName',2)
create(b,systemActorRealization,3)
addProperty(b,name, 'BName',4)
create(c,systemActorRealization,5)
addProperty(c,name, 'CName',6)
addReference(a,systemActorRealization,b,7)
addReference(a,systemActorRealization,c,8)
create(d,allocatedComponent,10)
addProperty(d,name, 'dName', 11)
addReference(b, allocatedComponent, d, 12)
addReference(c, allocatedComponent, d, 13)
  
```

**Fig. 2.** Model construction operation sequence

Figures 1 and 2 represent the same model in the form of respectively a set of objects along with attributes and associations and a sequence of editing actions. Both represent the *LogicalActor*  $a$ , the *SystemActorRealizations*  $b$  and  $c$  and the *AllocatedComponent*  $d$ . They also represent the *name* attribute of each object. The actions in timestamps 1 and 2 create the  $a$  object and set its name. The actions 3 – 6 create the  $b$  and  $c$  objects and set their *name* attributes. The actions in timestamps 7 and 8 create the associations between  $a$ ,  $b$  and  $c$ . Finally, the actions in timestamps 10 – 13 create the  $d$  object and associate it as the *allocatedComponent* of  $b$  and  $c$ .

## 2.2 PraxisRules: The Consistency Rules DSL for Praxis

PraxisRules is a rule-based logical DSL used to define consistency rules over Praxis sequences. This language is able to represent constraints over the order of the actions in a sequence or over the configuration of objects implied by it.

The Java code snippet below represents a consistency rule over the model in Figure 1. This rule makes sure that every *ActorRealization* of a *LogicalActor* is an instance of *Actor*. This is done by navigating through a logical actor (represented by the *logicalActor* variable whose declaration is not shown), obtaining the list of its realizations (line 4) and iterating through it (lines 5–8) and checking if every realization has an associated allocated component that is an instance of *Actor* (lines 9–13).

```
1 /* Ensures that the Actor Realization of a Logical Actor always
2 *   realizes an Actor (at the system analysis level).
3 */
4   EList<SystemActorRealization> actorRealisations =
5       logicalActor.getSystemActorRealizations();
6   Iterator<SystemActorRealization> iterator =
7       actorRealisations.iterator();
8   while (iterator.hasNext()) {
9       SystemActorRealization next = iterator.next();
10      Component allocatedComponent =
11          next.getAllocatedComponent();
12      if (null == allocatedComponent ||
13          !(allocatedComponent instanceof Actor)) {
14          return createFailureStatus(ctx_p,
15              new Object[] { logicalActor.getName() });
16      }
17  }
```

The following PraxisRule code snippet illustrates the definition of the same rule under an operation based model representation strategy. This code defines a rule called `check_LogicalActor_ActorRealization` which detects logical actors realized by a system realization that is allocated to an object that is not an actor or that is not realized by any system actor realization.

First of all, notice that this rule defines a logical expression based on logical connectives (`and{}` for conjunction, `or{}` for disjunction and `not{}` for negation). Capitalized terms represent variables and terms starting with the `#` sign represent meta-classes or associations. Another important fact is that Praxis actions are used as predicates in this logical expression.

```
1 ["Ensures that the Actor Realization of a Logical Actor always
2   realizes an Actor (at the system analysis level)"]
3 public check_LogicalActor_ActorRealization(A, R)
4     <=>
```

```

4   and {
5       create(A, #LogicalActor),
6       addReference(A, #systemActorRealizations, R),
7       or {
8           not {addReference(R, #allocatedComponent, _)},
9           and {
10              addReference(R, #allocatedComponent, A),
11              create(A, #Actor)
12          }
13     }
14 }.

```

The most important difference between this rule and the previous one is that this one is not based on navigating through the objects in the current configuration, but in looking for actions in the sequence that represents the current model.

The main advantage of this kind of rule is that it is possible to identify which rules need to be rechecked (and which parameters need to be rechecked) by a simple inspection of the rules. For example, every time an action *addReference* for the association *systemActorRealization* is performed this rule needs to be rechecked, because if a new system actor realization is being added to a logical actor, it is necessary to verify if it does not violate this rule.

### 3 Case Study

The advantages of the operation based model representation presented in the last section have been empirically validated on randomly generated UML models in [12]. However they have never been investigated in real industrial models. That is then the main motivation for the present study.

This section is organized as follows: Section 3.1 details the industrial context in which this study has been realized; Section 3.2 lists its main objectives and planning. Finally, Section 3.3 describes the environment in which it was effectively performed and Section 3.4 discusses its results.

#### 3.1 Industrial Context

In order to build an architecture of a software intensive system, many stakeholders contribute to the description of the system architecture. Following a model-based engineering approach, the different stakeholders will use modelling tools to describe the architecture and analysis tools to evaluate some properties of the architecture.

Thales has defined a model-based architecture engineering approach for software intensive systems, the ARCADIA method [13]. It defines a model organization of five abstraction levels (viewpoints) for mainstream engineering and a set of others viewpoints for speciality engineering, depending typically on non-functional constraints applied on the system to be engineered. The views conforming to these viewpoints are used by different stakeholders during the system

definition process. Therefore, techniques and tools to manage the consistency of an information bulk made of several views on a system are necessary. The ARCADIA method adopts a viewpoint-based architectural description such as described in the conceptual foundations of ISO/IEC 42010, Systems and Software Engineering - Architecture Description [2].

This ongoing standard attempts to specify the manner in which architecture descriptions of systems are expressed. This standard provides key concepts and their relationships to document an architecture. Its key concepts (ArchitectureFramework, ArchitectureDescription, Viewpoint, View and Correspondence rule) are defined thanks to the conceptual model illustrated by the Figure 3. This conceptual model defines the semantics of the different concepts we overview here. An architecture description aggregates several Architecture Views. A view addresses one or more system concerns that are relevant to some of the system's stakeholders. A view aggregates one or more Architecture Models. Each view is defined following the conventions of an Architecture Viewpoint. The viewpoint defines the Model Kinds used for that view to represent the architecture addresses stakeholders' concerns.

As stated in this standard, in architecture descriptions, one consequence of employing multiple views is the need to express and maintain the consistency between these views. The standard introduces the Correspondence Rule concept that states a constraint that must be enforced on a correspondence to express relation between architecture description elements (Views, Architectural Model, etc.). Correspondences can be used to express consistency, traceability, composition, refinement and model transformation, or dependencies of any type spanning more than a single model kind.

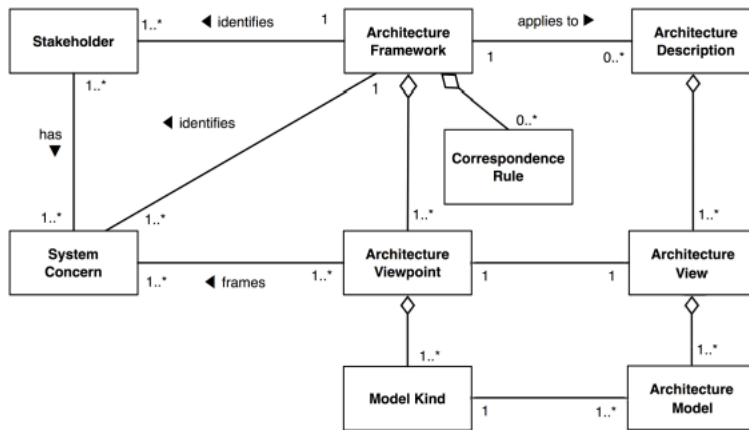


Fig. 3. ISO-IEC 42010 Architecture Framework overview

Considering this industrial context, it can be considered that there are 3 major types of model coherency to be managed:

- The first one aims at ensuring that a model conforms to its metamodel, i.e. that it addresses the well-formedness of the model. Since the modeling environment is DSL based (i.e. not profile based using a general purpose language), the well-formedness can be de facto ensured.
- The second one aims at ensuring that a model conforms to a coherent set of engineering rules; i.e. that the engineer conforms to a defined engineering method; in order to capitalize and reuse standard and domain specific engineering best practices.
- The third one aims at ensuring information consistency between distributed engineering environments, i.e. when there is not a unique centralized data reference. The main purpose here is to ensure coherency of all engineering activities across engineering domains, typically mainstream architecting and speciality engineering activities.

### 3.2 Objectives & Planning

Our experimentation focused on the second type of model coherency which consists in determining if a given configuration of set of views (models) are coherent with a set of consistency rules or not. This study consisted in detecting the inconsistencies between views conforms to the engineering meta-model defined by Thales and composed of a set of 20 meta-models and about 400 meta-classes involved in the five viewpoints defined in ARCADIA. The purpose of this study was assessing the benefits of Praxis Rules over Praxis strategy when compared to the traditional Java over EMF one. In terms of benefits, we study the efficiency to compute the set of inconsistencies in a given model and the usability of the approach.

In order to evaluate the effectiveness of the operation-based approach, we have validated the approach against our case study and the experiment environment with one Prolog expert and two Java developers from Thales. A set of 30 existing consistency rules initially implemented in Java over EMF have translated in Praxis Rules thanks to the praxis rule editor. We validated the consistency engine with models coming from operational contexts. We used three different models (ranging from 1.000 to 50.000 model elements) to determine the performance of the consistency engine tool for different model sizes.

### 3.3 Environment

Our experiment environment consisted of the Praxis consistency engine and a System engineering tool dedicated to this industrial context. This latter tool has been built on the top on the Eclipse Obeo Designer tool<sup>6</sup> and exposes a dedicated engineering language providing user-friendly ergonomics.

---

<sup>6</sup> <http://obeo.fr/pages/obeo-designer>

It allows engineers to define the architecture description of a software system by providing the five following views:

- The “Operational Analysis” model level, where the customer needs are defined and/or clarified in terms of tasks to accomplish by the System/Software, in its environment, for its users.
- The “System Analysis” model level, that provides a “black box” view of the System where the System limits are defined and/or clarified in terms of actors and external interfaces, the System capabilities and functional and non-functional needs and expectations; allowing to identify the more constraining/impacting requirements.
- The “Logical Architecture” model level, which provides a “white box” view of the System. It defines a technical and material independent decomposition of the System into components, and where the non-functional constraints are refined and allocated.
- The “Physical Architecture” model level, which is defined by the structuring architecture of the System. It takes into account non-functional constraints, reuses legacy assets and applies product policies.
- The “EPBS (End Product Breakdown Structure)” model level is an organizational view identifying the configuration items for development contracts and further Integration, Verification and Validation.

The Praxis Consistency engine has been integrated on top of this tool. It has been written in Java and is coupled with SWI-Prolog. From any given model, a equivalent sequence of editing operations is generated and passed to SWI Prolog. The Prolog engine then executes a set of queries representing consistency rules (also described in Prolog) and returns the list of detected inconsistencies to the user.

In the point of view of users, the Praxis Consistency engine provides two main components or features: the ConsistencyRule editor and the Consistency View. The first one allows the description of consistency rules using the PraxisRules DSL. These rules are then compiled into Prolog and are used by the Consistency engine. The Consistency View shows the number of inconsistencies found in the model and the model information that are not conform to the engineering rules.

A screen shot of the integrated tool is displayed in Figure 4. It shows an architecture description model that is being edited by an engineer (on the top) and the set of detected inconsistencies (the list on the bottom right). The only modification on this tool needed to this experiment was the inclusion of a timer that precisely indicated the time needed to perform each consistency verification.

### 3.4 Results & Evaluation

This section details the results of our experiments and the evaluation of the usability and applicability of Praxis in the industrial context under study. This section is divided into three parts, in the first one we analyze the comparison of the performance results obtained by Praxis and EMF. In the second part we



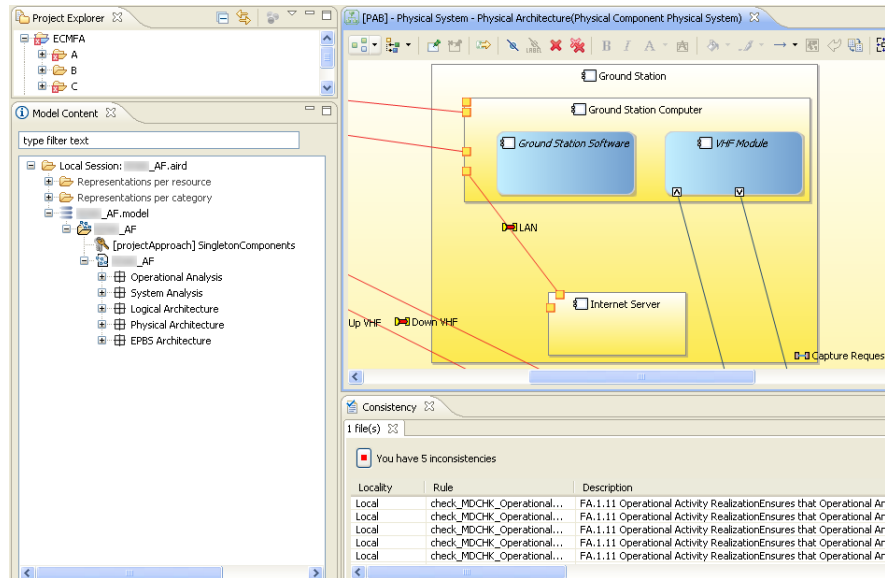


Fig. 4. Experiment Environment

analyze the adaptation of the rules written in the first part to the incremental checking model provided by Praxis. Finally, the third part describes our overall evaluation about the difficulty of rewriting part of our existing Java consistency rules in PraxisRules.

**Detecting model inconsistency** Table 1 describes the different metrics of the models used in our experiment. The model A is a toy model provided with the environment and the two others (B and C) are realistic models. The first three lines describe respectively the number of model elements in each model; the size of the model as represented as an EMF model and as a Praxis sequence of actions. Notice that the Praxis representation is in average four times more verbose than the EMF one. That happens because much more information is stored in Praxis, namely the order and the time-stamp of execution of each action.

The fourth, fifth, sixth and seventh lines display respectively the time needed to generate the actions file from the EMF model; the time needed to load it along with the translated consistency rules into the SWI Prolog engine and the amount of memory in MB necessary for the Eclipse process and the Prolog process to open each representation of it. These memory related numbers have been obtained by subtracting the amount of memory used by the process before and after loading the model. The amount of memory used before loading the EMF file averaged in 261 MB and before loading the Praxis sequences of actions averaged in 6MB.

Finally, the two last lines compare the time taken by each consistency engine to verify each model. Notice that, the “Java Overall check time” is always lesser than the “Praxis Overall check time”. Nevertheless, we consider that the ”Praxis Overall check time ” is acceptable for the set of rules in this context.

	Model A	Model B	Model C
<b>Number of model elements</b>	986	48 616	52 703
<b>EMF model file size (KB)</b>	326	11 282	13 724
<b>Praxis action file size (KB)</b>	1 398	37 589	66 562
<b>Time to generate actions file (ms)</b>	265	7 625	9 610
<b>Loading time (ms)</b>	1 516	11 968	12 031
<b>eclipse.exe (MB)</b>	41	63	62
<b>plcon.exe (MB)</b>	0,6	85,4	89,6
<b>Praxis Overall check time(ms)</b>	63	1 172	1 640
<b>Java Overall check time (ms)</b>	13	292	332

**Table 1.** Experimentation metrics

As an overall evaluation, the main limitation of the Praxis consistency engine lies in the fact that it works with another model representation, a file containing the sequence of editing actions represented as Prolog facts. This file has to be generated and loaded into the Prolog engine every time the model has been modified. For the two realistic models (B and C), the time to generate the praxis actions file is about 7 to 10 seconds and the time taken to load this file is about 12 sec. The performance penalty thus induced means that it would be impractical to use Praxis in an industrial usage by generating the action file from scratch and to load the actions files for each check.

**Incremental detection of model inconsistencies** Since testing the incremental mode in EMF would require the adaptation of the existing Java rules to this mode we decided to evaluate only the usability of the Praxis consistency engine, without comparing it to EMF.

After having opened the model in incremental mode, the user needs to wait for a batch mode check that computes the initial set of inconsistencies in the model. From this point on, when any modification is made to the model the incremental check is executed, taking into consideration only the subset of the models and of the inconsistency rules that is concerned by the modification that was performed. We evaluated both the time necessary to perform the initial batch verification and the time needed to verify the increments. The performance obtained by the former was equivalent to the performance of the batch checker already displayed in Table 1. The performance for the later averaged in 100ms.

As an overall evaluation, we concluded that the time needed to perform the initial batch verification to be reasonable and comparable to the time already needed to load the model on Eclipse. With respect to the incremental checking

time, we considered it to be quite transparent and independent of the number of rules that need to be re-checked. We consider that the main drawbacks of the praxis approach can be mitigated with the incremental mode.

**Inconsistency rules** Thanks to the consistency rule editor, 30 rules have been written by two Java developers without knowledge of Prolog in the beginning of the study and one Logic Programming expert. Java and PraxisRules are languages built on very different paradigms. Java is imperative in nature, and querying or checking a model typically consists in iteratively navigating and inspecting the model elements with the explicit use of loops and collections. PraxisRules, being built on top of Prolog, inherits its declarative nature, which means that the typical querying or checking code consists of a pattern to be identified in the sequence of actions.

In the present study, 60% of the inconsistency rules were dedicated to verify some realization relationship between the five views (i.e. verifying if an element in a view correctly realizes other elements in other views). This kind of rule is easy to write with the PraxisRule syntax. Nevertheless, the rules that have been written by the Prolog expert were more efficient in terms of performance than the ones written by the Java programmers. 4 rules could not be written without the Prolog expert because they required to use of language constructs that were not available in the basic PraxisRules *library*. These constructs needed to be implemented in Prolog and added to that library by the Prolog expert. In spite of the language not being typed, the Consistency Rule editor helped to overcome this limitation by adding warnings to the references in the PraxisRule code that were not found of the imported metamodels.

As an overall evaluation, we considered that knowledge of Prolog is a very important prerequisite to write PraxisRules consistency rules. Furthermore, for more advanced cases it could become difficult to translate rules from Java to Praxis, especially for complex engineering rules which were considered to be hard to implement. This difference is counterbalanced by the fact that no rules needed to be rewritten in order to use Praxis on the incremental mode. Our evaluation is that for some classes of rules (like the ones detecting simple patterns between different views) it is worth to write the them in PraxisRules.

## 4 Conclusion

This paper described our experiences on the impact of the Operation based underlying model representation strategy to the efficiency of the inconsistency detection task. We use the Eclipse Modeling Framework (EMF) as the reference for our study, which consisted in testing Praxis, an operation based consistency management, by reimplementing a set of 30 consistency rules from the engineering meta-model defined by Thales and comparing the time necessary to compute its consistency with the time needed by EMF. This case study was executed in an industrial context, with non-UML models.

Our results show that, in terms of computation time, the operation based approach is not better than the object based one. The difficulty of adapting these rules to being used in the incremental verification was also compared. Our results show that the work necessary to write the consistency rules is reduced by the use of the operation based approach, since they usually do not need to be rewritten to work on incremental mode.

As future works, we intend to compare the Praxis incremental model with the EMF one. We also intend to repeat this evaluation in non-EMF contexts.

**Acknowledgments** This work was partly funded by ANR project MOVIDA under the convention N° 2008 SEGI 011.

## References

1. Selic, B.: The pragmatics of model-driven development. *IEEE Software* **20**(5) (2003) 19–25
2. International Organization for Standardization: ISO/IEC FCD 42010: Systems and software engineering - Architecture Description (07 2010)
3. OMG: Meta Object Facility (MOF) 2.0 Core Specification (January 2006)
4. Mens, T., et al.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: *Model Driven Engineering Languages and Systems*. Volume 4199 of LNCS., Springer (October 2006) 200–214
5. Balzer, R.: Tolerating inconsistency. *Proc. Int’ Conf. Software engineering (ICSE ’91)* **1** (1991) 158–165
6. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: *IN HANDBOOK OF SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING*, World Scientific 329–380
7. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logics to maintain consistency between UML models. In: *UML 2003 - The Unified Modeling Language*. Volume 2863 of *Lecture Notes in Computer Science.*, Springer (2003) 326–340
8. Elaasar, M., Brian, L.: An overview of UML consistency management. *Technical Report SCE-04-18* (August 2004)
9. Blanc, X., Mougnot, A., Mounier, I., Mens, T.: Detecting model inconsistency through operation-based model construction. In Robby, ed.: *Proc. Int’l Conf. Software engineering (ICSE’08)*. Volume 1., ACM (2008) 511–520
10. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering* **2**(1) (1992) 31–57
11. Egyed, A.: Fixing inconsistencies in UML design models. In: *Proc. Int’l Conf. Software Engineering (ICSE’07)*, IEEE Computer Society (2007) 292–301
12. Blanc, X., Mougnot, A., Mounier, I., Mens, T.: Incremental detection of model inconsistencies based on model operations. In: *Proceedings of the 21st International Conference on Advanced Information Systems, CAISE’09*, Springer (2009) 32–46
13. Voirin, J.L.: Model-driven architecture building for constrained systems. *CSDM’10* (2010)