Detecting Model Inconsistency through Operation-Based Model Construction

Xavier Blanc, Isabelle Mounier, Alix Mougenot
LIP6
104 av. Président Kennedy
75016 Paris, France
+33 1 44 27 73 22
firstname.name@lip6.fr

Tom Mens
UMH
6 av. du Champ de Mars
7000 Mons, Belgique
+32 65 37 3453
tom.mens@umh.ac.be

ABSTRACT
Nowadays, large-scale industrial software systems may involve hundreds of developers working on hundreds of different but related models representing parts of the same system specification. Detecting and resolving structural inconsistencies between these models is then critical. In this article we propose to represent models by sequences of elementary construction operations, rather than by the set of model elements they contain. Structural and methodological consistency rules can then be expressed uniformly as logical constraints on such sequences. Our approach is meta-model independent, allowing us to deal with consistency between different models whatever their kind. We have validated our approach by building a Prolog engine that detects violations of structural and methodological constraints specified on UML 2.1 models and requirement models. This engine has been integrated into two contemporary UML-based modelling environments, Eclipse EMF and Rational Software Architect (RSA).

Categories and Subject Descriptors
D.2 [Software Engineering]: Design Tools and Techniques

I.6 [Simulation and Modeling]: Model Development; Model Validation and Analysis

General Terms
Design, Verification.

Keywords
Model, Meta-model, Consistency, Logic

1. INTRODUCTION
The more complex a system is, the more its development requires a collection of different models. Large-scale industrial software systems may involve hundreds of developers working on hundreds of different but related models representing parts of the whole system specification [16]. Ensuring consistency between those models becomes critical as even a minor inconsistency can lead to serious faults in the system and be the source of project failure.

In [6] and [7], Finkelstein *et al.* define an approach, called the *Viewpoints Framework*, where each developer has her own viewpoint composed only of models that are relevant to her. The main insight is that model consistency cannot and should not be preserved at all times between all viewpoints. The *Viewpoints Framework* proposes to allow for temporary model inconsistencies rather than to enforce model consistency at all times [2]. Spanoudakis and Zisman [17] define six inconsistency management activities that should be undertaken. The first activity, inconsistency detection, is of special interest as it defines the foundation of the whole process. Considering this activity, two families of approaches are identified: the logic-based approaches and the model checking approaches. The logic-based approaches are defined by the use of some formal inference techniques to detect any kind of model inconsistency. The model checking approaches deploy dedicated model verification algorithms that are well suited to detect specific behavioural inconsistencies but are not well adapted to other kinds of inconsistencies.

With the appearance of MDA [11], models are now more formally defined as their structures are specified conforming to a meta-model. A review of UML-based approaches dealing with model consistency is presented in [5]. It should be noted that most approaches follow the idea presented in [2][6] and fall into the logic-based approach family of [17] for addressing model inconsistency. However, their two major drawbacks are, firstly, that they only target structural constraints, and secondly, that they only deal with UML models.

In this paper, we argue that structural and methodological inconsistency detection should be supported uniformly. Methodological rules constrain the overall construction process, whereas structural rules only constrain the model obtained at the end of this process [17]. Regarding the model life cycle, those two kinds of rules are complementary as structural rules constrain model states whereas methodological rules constrain model changes. We stress the fact that, to our knowledge, no existing approach allows to define methodological consistency rules.

Moreover, we also argue that inconsistency detection should be supported uniformly for any model regardless its meta-model. On the one hand, complex models may become very large, requiring the need to fragment them in different smaller pieces. On the other hand, a wide variety of models is needed for building large-scale industrial software systems. Therefore, inconsistency rules

ICSE'08, May 10-18, 2008, Leipzig, Germany

should be considered as multi-model rules as they may target several meta-models and may have to be detected on sets of models.

The key idea of the approach that we present in this paper to uniformly detect structural and methodological inconsistencies is that it relies on elementary model construction operations instead of the model elements themselves. Any model, regardless its meta-model, can be represented as a sequence of operations performed to construct it rather than by the set of model elements it contains. Based on this key idea, both structural and methodological rules can be expressed uniformly as logical formulae on the sequence of construction operations.

The remainder of this paper is structured as follows. Section 2 presents the foundation of our approach, illustrating the definition and detection of structural and methodological inconsistencies on model construction sequences. Section 3 presents the prototype we built and section 4 presents the validation of our approach. Related work is given in section 5, and we conclude in the last section.

2. DETECTION OF INCONSISTENCIES BASED ON MODEL CONSTRUCTION

2.1 Model construction

A model is defined by a set of model elements, which are instances of meta-classes [12]. Independently from meta-models, a model can be considered as a set of model elements that own values and refer to each other. Each model element is an instance of a meta-class that defines the properties it can own and the references it can have [12]. Every model can be expressed as a sequence of elementary construction operations. The sequence of operations that produces a model is composed of the operations performed to define each model element. The four elementary operations we defined are inspired from the MOF reflective API [12]:

- *create*(*me*,*mc*) corresponds to the creation of a model element instance *me* of the meta-class *mc*. A model element can be created if and only if it does not already exist in the model. A newly created model element doesn't own values and doesn't refer to any other model element;
- *delete*(*me*) corresponds to the deletion of the model element instance *me*. A model element can be deleted if and only if it exists in the model and it is not referenced by any other model element. If it is, these references should be removed first using *setReference*;
- *setProperty(me,p,Values*) corresponds to the assignment of a set of *Values* to the property *p* of the model element *me*. Values can be assigned to the property *p* of a model element *me* if and only if *me* exists in the model, it is an instance of a meta-class *mc*, the property *p* is defined in *mc* or in a meta-class it inherits from, *p* and the assigned values have the same type, and the set of *Values* is consistent with the multiplicity of *p*;
- *setReference(me,r,References*) corresponds to the assignment of *References* to the reference *r* of the model element *me*. A set of model elements references can be assigned to the reference *r* of a model element if and only if the model element *me* exists in the model, it is an instance of a meta-class *mc,* the reference *r* is defined in *mc* or in a meta-class it inherits from, *r* and the assigned references have the same type and correspond to created model elements, and the set of *References* is consistent with the multiplicity of *r*.

Figure 1 presents a simplified part of the UML 2.1 meta-model that concerns the use cases [13]. It will be referred to as UCMM for Use Case Meta Model in the remainder of this article.

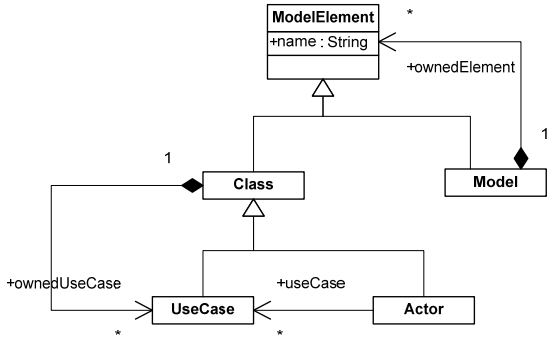

Figure 1. **The UCMM meta-model.**

Figure 2 represents a model instance of the use case part of the UCMM meta-model. This model is used as a running example to demonstrate our approach. It is composed of an actor (named "Customer") that is associated with three use cases (named "Create eBasket", "Buy eBasket" and "Cancel eBasket") belonging to a class (named "PetStore") representing the system.

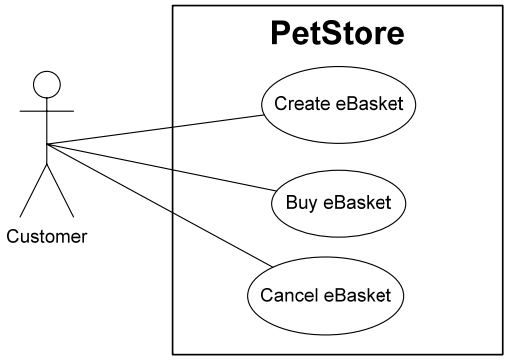

Figure 2. **A use case model instance of UCMM.**

The following construction sequence σ_{uc} can be used to produce the model of Figure 2:

1. create(c1,Class)
2. setProperty(c1,name, {'PetStore'})
3. create(uc1,UseCase)
4. setProperty(uc1,name, {'Buy eBasket'}))
5. create(uc2,UseCase)
6. setProperty(uc2,name,{'Create eBasket'})
7. create(uc3,UseCase)
8. setProperty(uc3,name,{'Cancel eBasket'})
9. setReference(c1,ownedUseCase,{uc1,uc2,uc3})
10. create(a1,Actor)
11. setProperty(a1,name, {'Customer'})
12. setReference(a1, usecase, {uc1,uc2,uc3})

Operation 1 corresponds to the creation of the class that owns use cases. Operations 3, 5 and 7 correspond to the creation of the three use cases. Operations 4, 6 and 8 correspond to the assignment of the name of the three use cases. Operation 9 links the three use cases with the class. Operations 10 and 11 correspond to the creation of the actor and to the assignment of its name. Operation 12 links the actor with the three use cases. This arbitrary sequence is used in the next sections to exhibit our consistency rules examples.

Figure 3 presents our proposal for a requirements meta-model. Note that this meta-model is only used here to exemplify our approach; it should not be considered as a contribution to the requirement management domain. Our requirements meta-model defines concepts usually used for elaborating requirements specifications. A requirements specification (Specification meta-class) is composed of two parts: issues (IssueSet and Issue meta-classes) and term definitions (Glossary and Entry meta-classes). An issue has a priority (low, medium, high), an id, a status (Status enumeration), a title and a definition. An issue refers to term definitions.

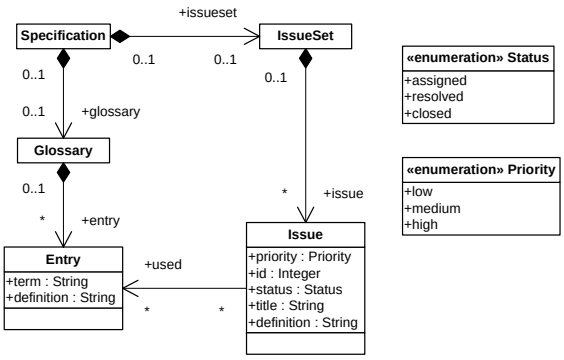

Figure 3. **Requirements meta-model**

The following sequence σ_{req} produces an example of a model instance of the requirements meta-model. It is composed of one issue (id 1253345) that makes use of two term definitions (e1: 'Customer' and e2: 'Basket'). This issue defines that a customer of the PetStore system has to either buy his opened eBasket or to cancel it before opening a new one:

1. create(spec,Specification)
2. create(is,IssueSet)
3. setReference(spec,issueset,{is})
4. create(gl,Glossary)
5. setReference(spec,glossary,{gl})
6. create(e1,Entry)
7. setProperty(e1,term, {'customer'})
8. setProperty(e1,definition, {'human connected and authentified. He/she has been registered in the PetStore system.'})
9. create(e2,Entry)
10. setProperty(e2,term, {'eBasket'})
11. setProperty(e2,definition, {'electronic representation of a classical basket'})
12. setReference(gl,entry,{e1,e2})
13. create(iss1,Issue)
14. setProperty(iss1,title, {'only one opened eBasket per customer'})
15. setProperty(iss1,id, {'1253345'})
16. setProperty(iss1,definition, {'If a customer has an opened eBasket, he/she has to either cancel it or buy it before opening another one'})
17. setProperty(iss1,priority, {'low'})
18. setProperty(iss1,status, {'assigned'})
19. setReference(iss1, used, {e1,e2})
20. setReference(is, issue, {iss1})

2.2 Detection of Inconsistencies

Several classifications of consistency rules have been provided in the literature [17][18][5]. Our goal is not to define yet another consistency rule classification but rather to propose a uniform mechanism for dealing with model inconsistency whatever the kind of consistency. For the sake of simplicity, we only consider two kinds of inconsistencies in this article: structural and methodological inconsistencies.

Structural consistency rules define relationships that should hold between model elements regardless of the way they have been constructed. These rules can be compared with well-formedness rules of [17], structural rules of [18] and syntactic, static rules of [5]. **Methodological consistency** rules are constraints over the construction process itself. They impose a certain order of operations that should be respected by the different developers that edit the models. These rules can be compared with development process compliance rules of [17].

For both structural and methodological consistency rules, we propose to define them as logic formulae over the sequence of model construction operations. For structural consistency rules, constraints mainly target operations of the sequence that impact the resulting model. For methodological consistency rules, formulae mainly target the order between operations of special interest (key operations of the construction process).

Using predicate logic has as main advantage to be based on well-defined sound semantics. Moreover, existing logical inference engines such as SWI-Prolog can be used. In the related work section, the advantages and limitations of logic reasoning compared to other approaches will be discussed.

The following subsections present representative examples of both structural and methodological inconsistency rules. For both kinds of inconsistency we present three examples, one for each meta-model (UCMM and Requirement) and one that spreads across both meta-models.

2.3 Structural Inconsistency

Structural inconsistency rules only concern the model obtained after executing the sequence of construction operations. Thus, only operations of the sequence that really impact the resulting model have to be taken into account while evaluating structural inconsistency rules. In other words, operations whose effects are cancelled by following operations in the sequence should not be taken into account.

More precisely, the following rules apply in order to identify operations whose effects are not cancelled:

- *create(me,mc)* has an effect if there is no following *delete(me)* operation appearing later in the sequence. This assures that the created model element *me* is not deleted.
- *setProperty(me,p,Values)* has an effect if there is no following operation in the sequence that is either a *setProperty(me,p,OtherValues)* operation targeting the same model element and the same property or a *delete(me)* operation targeting the same model element. This assures that the values are never changed and the model element is not deleted.
- *setReference(me,r,References)* has an effect if there is no following operation in the sequence that is either a *setReference(me,r,OtherReferences)* operation targeting the same model element and the same reference or a *delete(me)* operation targeting the same model element. This assures that the references are never changed and the model element is not deleted.
- *delete(me)* should never be considered while evaluating structural inconsistency rules as it only cancels the effect of other operations.

In the remainder of the paper, we will use operations such as *lastCreate*, *lastSetProperty* and *lastSetReference*. The '*last'* prefix indicates that those operations are not followed by operations cancelling their effects, as explained in the above. Structural inconsistency rules therefore only use those '*last'* operations.

2.3.1 UCMM example

The structural inconsistency example we choose on UCMM is inspired from the UML 2.1 specification. It specifies that an actor should not own a use case even if the meta-model permits it.

Operations concerned by this rule are creation of actors (*lastCreate(me,Actor)*) and assignment of owned use cases to them (*lastSetReference(me,ownedUseCase,References)*). A construction sequence produces a consistent model if and only if it does not assign owned use cases to actors. In other words, for all created actors (*lastCreate(me,Actor)*) there should be no assignment of owned use cases (*lastSetReference(me,ownedUseCase,R)*). Formally, this rule can be expressed by the following logical formula:

ActorsDoNotOwnUseCase (σ) = true iff
$\forall$ a $\in \sigma$,
if a = lastCreate (me_{ac}, Actor) then
$\nexists$ o $\in \sigma$,
o = lastSetReference(me_{ac},ownedUseCase, R)
and R $\neq \emptyset$.

Regarding this structural inconsistency rule, sequence σ_{uc} presented in section 2.1 produces a consistent model. Indeed, it contains only one *lastCreate(me,Actor)* (line 10) and no *lastSetReference(me,ownedUseCase, R)* for this actor.

2.3.2 Requirements example

An example of a structural inconsistency for the requirements meta-model is based on our own experience in requirements management. An issue should not use an entry if the issue's definition does not mention the entry's term.

Operations concerned by this rule are creation of issue (*lastCreate(me,Issue)*), assignment of their definition (*lastSetProperty(me,definition,Values)*) and assignment of their used entries (*lastSetReference(me,used,References)*). A construction sequence produces a consistent model if and only if for all created issues, the term of their assigned entries is mentioned in their definition. Formally, this rule can be expressed by the following logical formula[1]:

IssuesDoNotUseUselessEntry(σ) = true iff
$\forall$ a $\in \sigma$,
if a = lastCreate (me_{is}, Issue) and
$\exists$ u $\in \sigma$, u = lastSetProperty(me_{is},definition, DefVal) and
$\exists$ v $\in \sigma$, v = lastSetReference(me_{is},used, UE) with UE$\neq \emptyset$
then
$\forall$ me_{td} $\in$ UE
$\exists$ td $\in \sigma$, td = lastSetProperty(me_{td},term, TermVal) and
$\forall$ t $\in$ TermVal, $\exists$ d $\in$ DefVal, substring(t,d)

Regarding this structural inconsistency rule, sequence σ_{req} presented in section 2.1 produces a consistent model. Indeed, it contains only one *lastCreate(me,Issue)* (line 13), one *lastSetProperty(me,definition,DefVal)* (line 16) and one *lastSetReference(me,used,UE)* (line 19). Moreover, for each referenced entry e_1 and e_2, the sequence contains one *lastSetProperty(e,term,ValTerm)* (lines 7 and 10) that satisfies the substring comparison requirement in the last line of the formula: 'customer' and 'eBasket' are indeed substrings of 'If a customer has an opened eBasket,…'.

2.3.3 UCCM and Requirement example

Our example for a cross-model structural inconsistency aims at defining a dynamic (i.e., non-persistent) link between use case and requirements models. We express this dynamic link as follows: "each actor should have a corresponding entry in the glossary (i.e., the actor name and the term of the entry are the same)".

The particularity of this example is that it is based on two sequences, one for each model. Operations concerned by this rule are creation of actors (*lastCreate(me,Actor)*), assignment of their name value (*lastSetProperty(me,name,Value)*) and assignment of term values of entries (*lastSetProperty(me,term,Value)*). Two sequences produce a consistent pair of models if and only if for all created actors with an assigned name, there is a created term definition with a corresponding name value. Formally, this rule can be expressed by the following logical formula:

[1] The expression *substring(t,d)* in the last line of the formula is not an elementary construction operation, but simply a substring comparison predicate that is offered by the underlying logic programming language.

ActorNameCorrespondsToEntry (σ_{uc},σ_{rq}) = true iff

$\forall$ a $\in$ σ_{uc},

if a = lastCreate (me_{ac}, Actor) and

$\exists$ n $\in$ σ_{uc}, n = lastSetProperty(me_{ac},name, NameVal) then

$\exists$ t $\in$ σ_{rq}, t = lastSetProperty(me_{td},term, NameVal)

Regarding this structural inconsistency rule, the sequences presented in section 2.1 produce a consistent pair of models. Indeed, they contain only one *lastCreate(me,Actor)* (sequence σ_{uc}, line 10), one *lastSetProperty(me,name,NameVal)* (sequence σ_{uc}, line 11) and one *lastSetProperty(me,term,NameVal)* (sequence σ_{req}, line 7).

2.4 Methodological Inconsistency

As methodological inconsistency rules constrain the construction process, the order between operations should be taken into account during the evaluation of the rules. Therefore, in the rest of this paper we use the following notation:

Let $\sigma = a_1; \ldots; a_{m-1}; a_m; a_{m+1}; \ldots; a_n$ be a construction sequence. We denote by $a_i <_\sigma a_j$ the occurrence of operation a_i before a_j in sequence σ. We define the subsequences of σ preceding and following operation a_m as $Pred(\sigma,a_m) = a_1; \ldots; a_{m-1}$ and $Succ(\sigma,a_m) = a_{m+1}; \ldots; a_n$

2.4.1 UCMM example

An example of a methodological inconsistency on UCMM corresponds to the following methodological guidance: (1) Never unassign use case name; and (2) Assign a name just after the creation of a use case. Such guidance is automatically supported by nearly all UML case tool editors. Our framework provides more flexibility by allowing the user to define which methodological inconsistencies he needs, as well as when and how to enforce them.

Operations concerned by this rule are creation of use cases (*create(me,UseCase)*) and assignment of their name (*setProperty(me,name,Name)*). Indeed, a sequence is correct if and only if (1) there is no *setProperty(me,name,{""})* and (2) each *create(me,UseCase)* is immediately followed by a *setProperty(me,name,Name).* Formally, these rules can be expressed by the following logical formulae:

UCNaming1(σ) = true iff

$\forall$ a $\in$ σ, a $\neq$ setProperty(me,name,{""})

UCNaming2(σ) = true iff

$\forall$ a $\in$ σ, if a = create(me,UseCase) then

$\exists$ c $\in$ σ, c = setProperty(me,name, NameVal)

and $\nexists$ b $\in$ σ, a $<_\sigma$ b $<_\sigma$ c

Regarding these methodological inconsistency rules, sequence σ_{uc} presented in section 2.1 is consistent. Indeed, it contains no *setProperty(me,name,{""})* and the three *create(me,UseCase)* operations (lines 3, 5 and 7) are all immediately followed by the corresponding *setProperty(me,name,NameVal)* operation (lines 4, 6, 8).

2.4.2 Requirement example

A methodological inconsistency example for the requirement meta-model is a constraint on the issue life cycle. Indeed, in a requirements model, the status of an issue can only change in a predefined way. First, it must be set to "assigned", after which it can change to "resolved". Then it can change to "closed" after which it cannot be changed any longer.

Operations concerned by this rule are only the creation of issues (*create(me,Issue)*) and assignments of their status (*setProperty(me,status,Value)*). Indeed, a sequence is correct if and only if for all created issues (1) the first value of the status should be 'assigned'; (2) if the status is set to 'assigned' the next assignment has to be 'resolved'; (3) if the status is set to 'resolved' the next assignment has to be 'closed' and (4) if the status is set to 'closed' it cannot be changed anymore. Formally this rule can be expressed by the following mathematical formula:

IssueLifeCyle(σ) = true iff

$\forall$ i $\in$ σ, if i = create(me,Issue) then

(1) if $\exists$ snn $\in$ σ, snn = setProperty(me,status, val),
val$\neq${'assigned'} then
$\exists$ sr $\in$ Pred(σ,snn), sr = setProperty(me,status, {'assigned'})

(2) if $\exists$ sr $\in$ σ, sr = setProperty(me,status, {'assigned'}) then
$\forall$ snc $\in$ Succ(σ,sr), snc = setProperty(me,status, val)
and val$\neq${'resolved'}
$\exists$ sc $\in$ σ, sr $<_\sigma$ sc $<_\sigma$snc
and sc = setProperty(me,status, {'resolved'})

(3) if $\exists$ sr $\in$ σ, sr = setProperty(me,status, 'resolved') then
$\forall$ snc $\in$ Succ(σ,sr), if snc = setProperty(me,status, val)
then val={'closed'}

(4) if $\exists$ sr $\in$ σ, sr = setProperty(me,status, {'closed'}) then
$\nexists$ s $\in$ Succ(σ,sr), s = setProperty(me,status, val)

Regarding this methodological inconsistency rule, sequence σ_{req} presented in section 2.1 is consistent. Indeed, it contains a *create(me,Issue)* (line 13) and the first status assignment *setProperty(me,status,{'assigned'})* (line 18) is correct w.r.t. (1). Moreover, the sequence is also correct w.r.t. (2), (3) and (4) as there is no *setProperty(me,status, {'resolved'})* and no *setProperty(me,status, 'closed').*

2.4.3 UCCM and Requirement example

In section 2.3.3, the structural inconsistency rule we defined constrains actor names to be mentioned by entries in the glossary. The methodological inconsistency rule we chose constrains this link (between actors and entries) to be realized only after issues using the entries have been set to the 'assigned' status. Indeed, before having been set to the 'assigned' status, no developer is responsible of their realization. Therefore, no new actor corresponding to these issues should be created. In other words, new actors corresponding to issues can be created only after issues have been set to the 'assigned' status.

The particularity of this example is that it is based on two sequences, one for each model. Moreover, it defines order constraints between construction operations of both sequences. Therefore, a total order needs to be computed when evaluating the rule.

Operations concerned by this rule are creations of name assignments to actors (*setProperty(me,name,NVal)*) and term assignments to entries (*setProperty(me,term,TVal)*).

ActorNameWithEntriesOfAssignedIssues(σ) = true iff
  ∀ sn ∈ σ,
   if sn = setProperty(me,name,NVal) and
            ∃ a ∈ σ, a = create(me,Actor)
   then
    ∃ i, sr, st ∈ Pred(σ,sn),
      i = setProperty(mi,status,'assigned')
      sr = setReference(mi,used,Entries)
      st = setProperty(entry,term,NVal), entry ∈ Entries

When evaluating this methodological inconsistency rule, a total order has to be computed between the operations of the two sequences presented in section 2.1. The way in which the total order is computed will affect the result of this inconsistency rule. If we decide to put σ_{req} before σ_{uc} then the total ordered sequence will be consistent. However, if we decide to put σ_{uc} before σ_{req} then the total ordered sequence will be inconsistent. The other combinations depend on the order between the name assignment of the actor on one hand and the status assignment and its association to an entry on the other hand. In our particular example, this means that lines 7, 18, 19 of σ_{req} should precede line 11 of σ_{uc} for the total ordered sequence to be consistent.

3. REALIZATION

As a proof of concept of our approach, we have built a prototype in the logic programming language Prolog. The key idea is that inconsistency rules are translated to Prolog queries and model construction operations to Prolog facts. This Prolog engine has been integrated into the modeling environments Eclipse EMF and Rational Software Architect. Thanks to this integration, users can ask to the Prolog engine to perform the inconsistency check.

3.1 Architecture

Our prototype is composed of two main components: the *Sequence Builder* and the *Check Engine* (see Figure 4). The Sequence Builder is responsible for building a totally ordered sequence of construction operations from actions performed by developers, and storing it into a Prolog fact base. The Check Engine is responsible for detecting inconsistencies. It analyses the sequence computed by the sequence builder in the fact base and produces an inconsistency detection report.

3.2 Sequence Builder

We have defined two kinds of Sequence Builders. One is a *file reader* and the other is an *event listener*. The file reader can parse an XMI file containing a model and outputs the corresponding model construction sequence. The event listener can receive distributed events raised by developers while they edit their models in the Eclipse EMF editor or in Rational Software Architect (RSA). This enables the incremental checking of inconsistencies. A screenshot illustrating this approach is given in Figure 5. This screenshot shows a requirement model edited within Eclipse/EMF that is monitored by an *event listener* sequence builder.

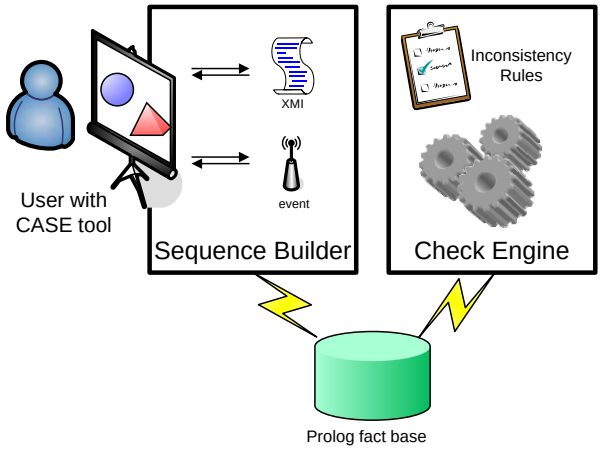

Figure 4. Tool architecture.

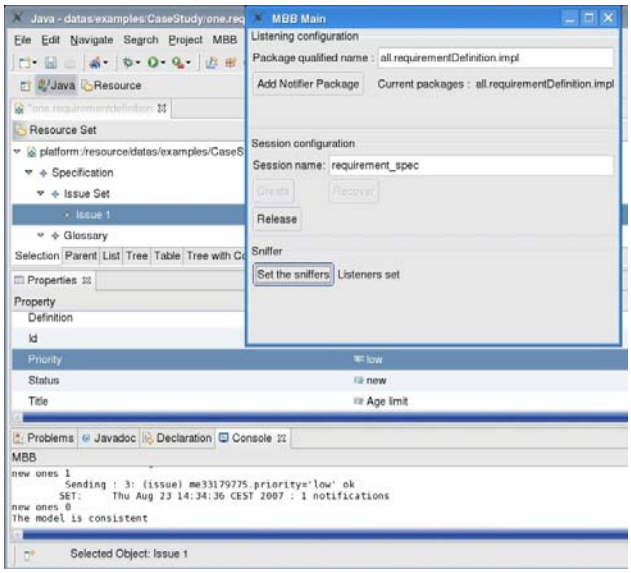

Figure 5. Screenshot of the event listener sequence builder integrated into the Eclipse EMF framework.

The *file reader* sequence builder has been developed in Java on top of the EMF framework. It opens an EMF XMI file containing the model and it traverses this model through the containment association (in EMF, all models are organized as trees thanks to this containment association). Whenever a new model element is visited, a corresponding *create* operation is appended to the model construction sequence and, for each of its properties, a corresponding *setProperty* operation is appended to the sequence. If the model element has references and if the referenced model elements have already been visited, a *setReference* operation is appended to the sequence; otherwise the reference is flagged in order to be appended as soon as the referenced model elements are visited. This algorithm always builds a *minimal* sequence of construction operations as there is no operation that cancels

previous operations in the sequence. Moreover, many different sequences can be obtained from a same model as the iteration is only based on the aggregation association and is therefore non deterministic.

We have used the EMF reflective API in order to parse all models regardless their metamodel. The file reader sequence builder was used to build construction operation sequences of huge UML 2.1 models (containing around 70000 model elements). The length of the resulting construction sequence exceeded 1 billion operations in some cases.

The *event listener* sequence builder has been developed in Java RMI on top of the EMF notification framework. It is composed of two subcomponents: the event sender and the event receiver components. This architecture has been defined to be deployed on distributed environments (with multiple distributed developers). However, we have only deployed and tested it so far with one event sender and one event receiver.

The event sender is an Eclipse plugin that makes use of the EMF notification framework (for UML 2.1, it makes use of the EMF transactional notification system). It is deployed on the developers Eclipse workbench. Each time a developer modifies his models, the event sender is notified and makes an RMI call to the event receiver. There is only one deployed event receiver that receives RMI calls made by the event senders. It builds a corresponding sequence of construction operations. Up till now, no special strategy has been elaborated to compute the total order; it is simply based on the order of receptions of RMI calls.

Both kinds of sequence builder output the computed sequence in a Prolog fact base. This fact base contains *create*, *delete*, *setReference* and *setProperty* facts corresponding to the operations presented in section 2. In addition to the arguments of the construction operations, these facts hold a time stamp that is used to define the total order.

3.3 Check engine

We chose to build our Check Engine on top of Prolog. Our approach consists in detecting elementary operations violating inconsistency rules within a model construction sequence. The idea is that, for a given inconsistency rule, a set of operations describes the cause of its violation. If such operations are detected within a sequence then the rule is violated. As we chose to encode elementary operations as facts, this approach fits the Prolog paradigm which is very efficient to gather facts. Moreover, a first diagnostic can be obtained automatically by returning all operations violating the inconsistency rules. Therefore, in our approach, an inconsistency rule is a Prolog query that detects operations causing inconsistencies. Such Prolog queries can be seen as negations of the formal consistency rules presented in section 2. We chose to express inconsistency rules using the full Prolog language (not limited to first-order logic), as it offers a lot of convenient facilities such as manipulation of primitive types (including lists) and result sets.

For structural inconsistency rules, we have defined queries corresponding to *lastCreate*, *lastSetProperty* and *lastSetReference* operations presented in section 2. Those queries make use of time stamps to return the *last* operation. Note that, *lastSetProperty* and *lastSetReference* will also return false if the third parameter of the *last* operation is an empty list. As an example, the following Prolog query corresponds to the UCMM structural inconsistency example of subsection 2.3.1, expressing that an actor should not own use cases:

```
analysis(X,Y) :-
  lastCreate(X,actor),
  lastSetReference(X,ownedusecase,Y).
```

This query computes all pairs (X,Y) where X is an actor and Y is a non-empty list of use cases owned by X. Prolog will find all X such that *lastCreate(X,actor)* is true in the sequence; therefore all actors present in the resulting model will be detected. For each identified actor, Prolog will evaluate whether *lastSetReference(X,ownedusecase,Y)* is true, implying that the actor references uses cases (Y) in the resulting model. If the query returns an X, then the rule is violated since there are actors in the resulting model that own use cases.

For methodological inconsistency rules, we have defined queries based on the time stamp for detecting if an operation occurs *before* or *after* another one within a given model construction sequence[2]. For example, the following Prolog query corresponds to the first part of the Requirement methodological inconsistency example rule of subsection 2.4.2, expressing that the first status assignment to an issue should be to the 'assigned' value:

```
requirementM(X,TSX) :-
  setProperty(X,status,Val,TSX),
  Val \= 'assigned',
  not((setProperty(X,status,'assigned',TSnew)
       ,before(TSnew,TSX))).
```

This query computes all pairs (X,TSX) where X is an issue and TSX a time stamp. Prolog will find all X such that *setProperty(X,status,Val,TSX)* is true in the sequence and where *Val* is different from 'assigned'; therefore all issues whose status is not set to the 'assigned' value will be detected. For each detected issue, Prolog will return those whose status has not been set to the 'assigned' value before. If the query returns an X, then the rule is violated since there are issues with status not initially set to the 'assigned' value. For those X, the time stamp of the status assignment violating the rule (TSX) will be returned.

4. Validation

Our prototype has been validated on the examples presented in section 2, and stress-tested on a real, large-scale UML model.

To achieve the latter, 58 UML 2.1 OCL constraints were translated into Prolog queries. We chose to not build an automatic transformation from OCL to Prolog queries. The reason was mainly because this would require us to formally define the semantic bridge between OCL (including the whole OCL library) and our formalism. We have translated "by hand" all the OCL constraints targeting the class diagram part of the UML 2.1 standard. For instance, there is an OCL constraint that defines that, for *Operations*, "*An operation can have at most one return parameter (i.e., an owned parameter with the direction set to 'return')*":

[2] This corresponds to the use of *Pred* and *Succ* in section 2.4.

In OCL, this constraint is expressed as follows:

```
Context Operation:
  ownedParameter -> select(par |
       par.direction = #return)->size() <= 1
```

We have translated this OCL constraint as follows:

```
operationOCL1(X) :-
  lastSetReference(
      X,ownedparameter,OwnedParameters),
  member(OP, OwnedParameters),
  lastSetProperty(OP,direction,'return'),
  member(OP2,OwnedParameters),
  OP2 \== OP,
  lastSetProperty(OP2,direction,'return').
```

This query computes all (X) where X is an operation that violates the inconsistency structural rule. Prolog will find all X such that *lastSetReference(X,ownedparameter,OwnedParameters)* is true in the sequence, therefore all operations in the resulting model will be detected. For each identified operation, Prolog will evaluate if it owns two different parameters (OP and OP2) that have a 'return' direction. If the query returns an X, then the rule is violated since there are operations in the resulting model that own at least two 'return' parameters.

A huge UML model was obtained by reverse engineering the Azureus project [1], which is known to possess a messy architecture. The model construction sequence for this UML model contained about 1.3 million construction operations. The engine needed about 45 seconds to load the entire model into memory and less than 3 minutes for checking the 58 consistency rules on this model. Only 9 consistency rules returned inconsistent model elements. Two of them returned 16000 model elements. This huge amount is probably due to the reverse engineering which certainly produces errors. The seven other rules return 50 inconsistent model elements which are probably due to human errors. These results are encouraging but cannot be considered as an evidence that our prototype can really manage models represented by billions of elementary operations. The validation test was done using SWI-Prolog version 5.6.32 for i386-linux, and SUN's HotSpot(TM) JVM build 1.6.0_01-b06. The machine used for the test is a i686 Bicore Intel(R) Pentium(R) D CPU 3.00GHz with 3 GB Ram memory running under Linux Kernel 2.6.17-5mdv.

Although 3 minutes seems quite a long time, the average time needed for checking a rule was less than 3 seconds, and 25 of the rules were checked in less than 100 milliseconds each. This time could be reduced even further by optimizing the Prolog rules. The constraint that was most expensive was the one to detect cycles in the containment relation. This rule, that did not reveal any inconsistency, required 2 minutes and 7 seconds to compute, thereby representing about 80% of the total checking time.

This test has also shown that it takes more time to prove that a rule is respected than finding the elements engaged in an inconsistency. This is mainly due to the way in which we have written the consistency rules, the checker will have to test all the possible elements and run all the test code before assessing that the rule is never violated.

5. RELATED WORK

The underlying idea to represent models as sequences of construction operations rather than a set of model elements is based on earlier work in software versioning systems. In particular, Lippe and Van Oosterom [8] proposed an operation-based approach to software merging. They showed that representing software versions as operation sequences significantly facilitates the process of detecting and resolving merge conflicts. In the future, we will exploit this benefit by applying our technique in the context of distributed modeling, where there is a need to merge parallel changes that have been performed to models in a distributed way.

In [15], Saito stressed that approaches based on elementary operations are the most adapted for large-scale environments. In a distributed large scale environment, only operations of interest, and not the complete model, have to be exchanged between pairs for checking inconsistencies. This reduces considerably the number of exchanged messages as well as their sizes.

OCL (Object Constraint Language) is the OMG standard that is mainly used to specify model structural inconsistency rules [14]. However, it is not possible to define methodological constraints with just OCL. In addition, OCL constraints are not allowed to have multiple contexts. Thus, they are not well suited to constrain a set of models.

In [9], a framework for inconsistency management was proposed. This framework also follows a logic-based approach but only for UML models. The approach is not based on construction operations but on model elements. It is not adapted to express methodological inconsistency rules. However, the framework not only offers inconsistency detection but also inconsistency resolution based on graph transformation. We think that our approach for detection of inconsistencies can be integrated with the proposed approach for inconsistency resolution and we envisage this integration as a short-term perspective.

In [10], another framework for inconsistency management has been proposed. This framework considers all resources as XML documents and proposes an extension to XPath for expressing consistency rules. Based on these rules, a checker can then detect inconsistencies between a set of distributed XML documents. This framework is well suited to detect structural inconsistency rules and does not seem to be applicable to methodological constraints.

One of the problems of logic programming concerns decidability. If the full power of Prolog is used, it is easily possible to run into infinite loops. It is our view that it is the responsibility of the developer of inconsistency rules to avoid such problems. The alternative would be to resort to a decidable variant of logic, such as description logics. This approach has been investigated, among others, in [18].

6. CONCLUSION

Both structural and methodological model inconsistency detection has to be performed on any model whatever its meta-model. This includes for instance, inconsistency detection between a requirement model and a design model in order to manage traceability. Inconsistencies have to be detected while developers elaborate the models in collaboration.

Our approach (1) is based on model construction operations, (2) uses logical constraints to define inconsistency rules and (3) is meta-model independent. We have shown in this article that structural and methodological inconsistencies can be expressed and verified in a uniform way using logical formulae. As highlighted with our examples, both intra-model and inter-model inconsistency rules can be defined and checked. Even if we have stress-tested our approach

only for UML models, it has been validated on different and independent meta-models as well.

It is worthwhile to note that, in our approach, all checks were executed in batch mode. The entire model was loaded in memory, and the rules were verified one after the other on the entire model. This is clearly not the most effective way to check model consistencies with our approach. An incremental consistency checking would be much more effective. It allows reusing results from previous checking phases, and requires testing only the necessary set of rules and elementary operations engaged in model modifications that occurred in between two consistency check phases. Egyed proposed a framework dedicated to instant inconsistency detection [3][4]. It dynamically instantiates particular rules each time an inconsistency is detected. Thanks to these rules, whenever an element is modified in the model, the framework is able to determine if some inconsistency rules are concerned by the change and need to be re-verified or not. In our approach, we are able to compute how rules access the sequence of elementary operations using the introspection facilities of Prolog. From this static knowledge it is possible to generate an incremental checker for a set of rules following the spirit of Eyged's framework. This incremental checker inputs one elementary operation, determines which inconsistency rules are concerned by this operation and, if any rule needs to be rechecked, an instantiated query call for this particular change is used to check if the rule is violated by this particular operation. This incremental checker is generated only once as the information needed to build it can be obtained in a static way from the set of inconsistency rules. We plan to experiment with the generation of the incremental checker for the 58 OCL constraints targeting the UML class diagrams

Finally, from its definition, our approach promises to scale up to inconsistency detection of large-scale distributed models as the detection only requires to exchange a limited number of operations as opposed to the complete model. A detailed analysis of this advantage, however, is left for future work.

7. ACKNOWLEDGMENTS

This work is supported in part by the IST European project "MODELPLEX" (contract no IST-3408) and by the FNRS FRFC project 2.4519.05 "Centre de Recherche en Restructuration Logicielle."

8. REFERENCES

[1] Azureus Java Open Source Project, http://azureus.sourceforge.net/

[2] Balzer, R. *Tolerating inconsistency*. In: Proc. Int'l Conf. Software Engineering (ICSE) , ACM (1991) 158–165

[3] Egyed, A. Instant Consistency Checking for the UML. Proc. Int'l Conf. Software Engineering (ICSE), pp. 381-390, ACM, 2006

[4] Egyed. A. Fixing Inconsistencies in UML Design Models. Proc. Int'l Conf. Software Engineering (ICSE), pp. 292-301, ACM, 2007

[5] Elaasar, M. Briand, L. *An Overview of UML Consistency Management*, Carleton Technical Report SCE-04-18, August 2004.

[6] Finkelstein, A. Spanoudakis, G. Till, D. *Managing Interference*, Joint Proc. Sigsoft `96 Workshops -- Specifications `96, ACM Press, pp. 172-174.

[7] Finkelstein, A. et al. *Inconsistency Handling in Multiperspective Specifications*, IEEE Transactions on Software Engineering, Vol.20, N°8, August 1994, pp569-577

[8] Lippe, E. Van Oosterom, N. *Operation-Based Merging*. Proc. ACM SIGSOFT Symp. Software Development Environments. 17(5). 1992, pp. 78-87

[9] Mens, T. Van Der Straeten, R. D'Hondt, M. *Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis*, Proc. Models 2006, Volume 4199 of Lecture Notes in Computer Science., Springer-Verlag (2006), pp. 200-214.

[10] Nentwich, C. Capra, L. Emmerich, W. Finkelstein, A. *xlinkit: a Consistency Checking and Smart Link Generation Service*. ACM Transactions on Internet Technology, 2 (2). pp. 151-185. ISSN 15335399.

[11] OMG, MDA Guide v1.0.1, 2003, omg/03-06-01

[12] OMG, MOF 2.0, 2006, formal/06-01-01

[13] OMG, UML 2.1 Superstructure, 2006, ptc/06-01-02

[14] OMG, OCL Object Constraint Language v2.0, formal/06-05-01

[15] Saito, Y. Shapiro, M. *Optimistic Replication*, ACM Computing Surveys, Vol. 37, No. 1, March 2005, pp. 42–81.

[16] Selic, B. *The Pragmatics of Model-Driven Development*, IEEE Software, Volume 20, Issue 5 (September 2003), pp. 19-24.

[17] Spanoudakis, G. Zisman, A. *Inconsistency Management in Software Engineering: Survey and Open Research Issues*, World Scientific Publishing, Handbook of SE & KE, Volume 1 (2001), pp. 329-380.

[18] Van Der Straeten, R. Mens, T. Simmonds, J. Jonckers, V. *Using description logics to maintain consistency between UML models*. Proc. UML 2003. Volume 2863 of Lecture Notes in Computer Science., Springer-Verlag (2003) pp. 326-340.