Incremental Detection of Model Inconsistencies Based on Model Operations

Xavier Blanc[1], Alix Mougenot[2,⋆], Isabelle Mounier[2], and Tom Mens[3,⋆⋆]

[1] INRIA Lille-Nord Europe, LIFL CNRS UMR 8022,
Université des Sciences et Technologies de Lille, France
[2] MoVe - LIP6, Université Pierre et Marie Curie, France
[3] Service de Génie Logiciel, Université de Mons-Hainaut, Belgium

Abstract. Due to the increasing use of models, and the inevitable model inconsistencies that arise during model-based software development and evolution, model inconsistency detection is gaining more and more attention. Inconsistency checkers typically analyze entire models to detect undesired structures as defined by inconsistency rules. The larger the models become, the more time the inconsistency detection process takes. Taking into account model evolution, one can significantly reduce this time by providing an incremental checker. In this article we propose an incremental inconsistency checker based on the idea of representing models as sequences of primitive construction operations. The impact of these operations on the inconsistency rules can be computed to analyze and reduce the number of rules that need to be re-checked during a model increment.

1 Introduction

Model driven development uses more and more complementary models. Indeed, large-scale industrial software systems are currently developed by hundreds of developers working on hundreds of models of different types (e.g. SysML, UML, Petri nets, architecture, work flow, business process) [1]. In such a context, model inconsistency detection is gaining a lot of attention as the overlap between all these models (that are often maintained by different persons) is a frequent source of inconsistencies.

Detection of inconsistencies was first introduced by Finkelstein et al. [2]. They defined the Viewpoints Framework, where each developer owns a viewpoint composed only of models that are relevant to him. The framework offers facilities to ensure consistency between viewpoints. The main insight is that model consistency cannot and should not be preserved at all times between all viewpoints [3]. The Viewpoints Framework suggests to allow for temporary model inconsistencies rather than to enforce model consistency at all times.

⋆ This work was partly funded by the french DGA.
⋆⋆ This work was partly funded by Action de Recherche Concertée AUWB-08/12-UMH19, *Ministère de la Communauté française, Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, Belgique.*

In all approaches that deal with detection of inconsistencies [4,5,6,7,8,9], the detection invariably consists in analysing models to detect inconsistent configurations defined by inconsistency rules. Therefore, the larger the models, the longer the detection process takes. Moreover, the large number of inconsistency rules and their complexity are two other factors that make the detection process highly time consuming. The impact of model changes should also be considered by consistency checkers. Indeed, developers keep modifying and improving their models, and some of these modifications may give rise to new model inconsistencies. Due to the time it takes, re-checking the entire model after each such model increment is unfeasible in practice.

This situation explains why there is an increasing focus on scalability issues [6,9,10]. The challenge is to check inconsistencies on large models continuously during their frequent evolution. As the detection of inconsistencies implies to find structures within a model, efforts mainly target the process of the incremental detection in its whole (what rules to check and when) and aim at not performing a complete check of the model each time it evolves.

In this article, we propose to address this challenge by providing an incremental inconsistency checker that only adds a small fixed amount of memory to run on top of our classical inconsistency checker. Given a model that has already been checked for inconsistency, and given a model increment (i.e., a sequence of modifications to this model), our goal is to identify those inconsistency rules that need to be re-checked. Section 2 explains how to detect inconsistencies and gives a formal definition of an incremental checker. Our proposal is based on the operation-based model construction approach presented in [9], which is briefly revisited in section 3. Section 4 presents our incremental checker, and section 5 provides a case study to validate our approach. Section 6 presents related work in this domain and we conclude in Section 7.

2 Detection of Inconsistencies

2.1 Inconsistency Rules

Detection of inconsistencies consists in analyzing models to identify unwanted configurations defined by the inconsistency rules. If such configurations are found in the model, then the model is said to be inconsistent. Inconsistency rules can be compared to the negation of well-formedness rules of [11], structural rules of [12] and syntactic rules of [13].

One can see an inconsistency check as a function that receives as input a model and a set of inconsistency detection rules and that returns the evaluation result of each rule. If a rule evaluates to true (i.e., the model is inconsistent), then the model elements causing the inconsistency are also returned by the check function.

In this paper we use two inconsistency examples that are inspired by the class diagram part of the UML 2.1 specification [14]. Figure 1 presents a simplified fragment of the UML 2.1 meta-model for classes [14]. It will be referred to as CMM for Class Meta Model in the remainder of this article.

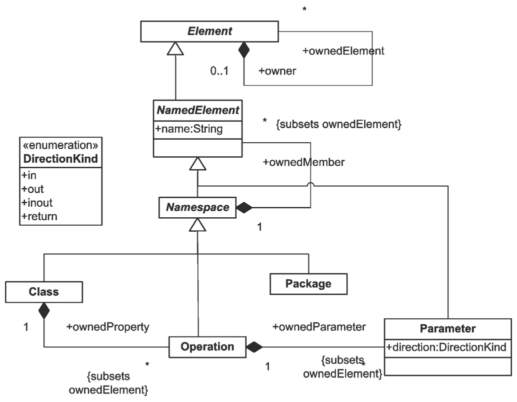

Fig. 1. CMM: A simplified fragment of the UML 2.1 meta-model

The two inconsistency rules we use are specified in the UML 2.1 specification: `OwnedElement` defines that an element may not directly or indirectly own itself; `OwnedParameter` defines that an operation can have at most one parameter whose direction is 'return'.

Figure 2 shows a model instance of the CMM meta-model. This model is used as a running example to illustrate our approach. It is composed of a package (named 'Azureus') that owns two classes (named 'Client' and 'Server'). The class 'Server' owns an operation (named 'send') that does not own any parameter. The model is consistent w.r.t. our two inconsistency rules.

2.2 Incremental Checking

During any model-driven software project, models are continuously modified by developers. As each modification can impact many model elements, checks should be performed as often as possible during the development life cycle in order to have a good control over the model consistency. However, as the time needed to perform a check can be very high, the challenge is to control efficiently the consistency of the model without burdening or delaying the developer in his other modeling activities.

One way of dealing with this problem is to provide an incremental checker. Incremental checks take into account the modifications made to a model. Rather than re-checking the entire model, one can analyze the impact of a set of modifications (the model increment δ) on the consistency of a model, and only re-check those inconsistency rules whose value may potentially have changed. In this way, the number of rules to be checked after modification may be reduced significantly, leading to an increased performance of the algorithm when compared to checking the inconsistency of the entire model.

Ideally, for an incremental check to be efficient, two considerations should be made. First, an inconsistency rule should be incrementally re-checked only if the modifications contained in the model increment change its previous evaluation.

Second, after a modification, only the part of the model that is concerned by the modifications needs to be analyzed in order to perform the new evaluation.

Following those two considerations while building an incremental checker, the incremental check should (1) evaluate only a subset of inconsistency rules and (2) analyze only a subset of model elements. Currently our approach only targets the first point and aims at filtering at a low level of granularity those rules that need to be re-checked after some model increment. It should be noted that our approach only needs a small fixed memory size to run on top of our classical inconsistency checker.

3 Detection of Inconsistencies Based on Model Construction

3.1 Operation-Based Model Construction

In [9], we propose to represent models as sequences of elementary operations needed to construct each model element. The four elementary operations we defined are inspired by the MOF reflective API [15]:

1. **create***(me,mc)* creates a model element *me* instance of the meta-class *mc*. A model element can be created if and only if it does not already exist in the model;
2. **delete***(me)* deletes a model element *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;
3. **setProperty***(me,p,Values)* assigns a set of *Values* to the property *p* of the model element *me*;
4. **setReference***(me,r,References)* assigns a set *References* to the reference *r* of the model element *me*.

Figure 3 is the construction sequence σ_c used to produce the model of Figure 2. In Figure 3, line 1 corresponds to the creation of the package; line

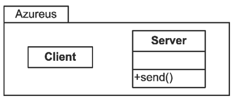

Fig. 2. Azureus UML model

```
1  create(p1,Package)
2  setProperty(p1,name, {'Azureus'})
3  create(c1,Class)
4  setProperty(c1,name, {'Client'}))
5  create(c2,Class)
6  setProperty(c2,name,{'Server'})
7  setReference(p1,ownedMember,{c1,c2})
8  setReference(p1,ownedElement,{c1,c2})
9  create(o1,Operation)
10 setProperty(o1,name, {'send'})
11 setReference(c2, ownedProperty, {o1})
12 setReference(c2, ownedElement, {o1})
```

Fig. 3. Model construction operation sequence σ_c

2 corresponds to the assignment of the name of the package; lines 3 and 5 correspond to the creation of the two classes; lines 4 and 6 to the assignment of the name of the two classes; line 7 links the two classes to the the package's owned members; line 8 does the same but with the owned element list of the package (the parameter list subsets the element list); lines 9 and 10 correspond to the creation of the operation and its name affectation; line 11 links the operation to the class' properties; line 12 does the same but with the element list of the class (the property list subsets the element list). This arbitrary sequence is used in the next sections to illustrate our incremental inconsistency checker.

3.2 Inconsistency Detection Rules

Our formalism allows to define any inconsistency rule as a logic formula over the sequence of model construction operations. As syntactic shortcut, we define the '`last`' prefix to denote operations that are not followed by other operations canceling their effects. For instance, a `lastCreate(me, Class)` operation is defined as a `create(me, Class)` operation that is not followed by a `delete(me)` operation; and a `lastSetReference(me, ownedProperty, val)` operation is defined as a `setReference(me, ownedProperty, val)` operation for which the value of the ownedProperty reference of `me` in the model corresponds to `val`. A complete description of the semantics of the '`last`' operations is provided in [9].

For the `OwnedProperty` inconsistency rule, the operations that can make a model inconsistent are the ones that modify a reference to the ownedParameter list of an operation and the ones that modify the direction of a parameter. More formally, those operations are `setReference(me,ownedParameter,`θ`)` where $\theta \neq \emptyset$ and `setProperty(me,direction,val)`.

This inconsistency rule can be formalised as follows:

$$\begin{aligned} &OwnedParameter(\sigma) = \{me \mid \exists sr, sp1, sp2 \in \sigma \cdot \\ &\quad sr = setReference(me, ownedParameter, \theta) \wedge \\ &\quad sp1 = setProperty(p1, direction, `return') \wedge \\ &\quad sp2 = setProperty(p2, direction, `return') \wedge \\ &\quad p1, p2 \in \theta \wedge p1 \neq p2\} \end{aligned}$$

Sequence σ_c of Figure 3 produces a model that is consistent with rule `OwnedParameter`, as it contains only one operation (line 9) that has no parameter.

For the `OwnedElement` inconsistency rule we presented in section 2.1, the only operation that can make a model inconsistent is the one that adds a reference to the ownedElement list of a model element. More formally, this operation is `setReference(me, ownedElement,` θ`)` where `me` is an element and θ is not empty. Such an operation produces an inconsistent UML model if and only if the set θ is such that a cycle appears among the `ownedElement` references. The way to repair such an inconsistent model is to break the cycle by removing a relevant reference. One can easily check that sequence σ_c produces a model that is consistent with rule `OwnedElement`.

4 Incremental Checking Based on Model Operations

Our incremental inconsistency checker reduces the set of inconsistency rules that need to be re-checked. Our approach is based on analyzing the impact that operations of the model increment may have on the evaluation of the inconsistency rules. We define a partition of equivalence classes for construction operations and use this partition to classify the inconsistency rules. Section 4.1 presents the equivalence classes and section 4.2 explains how those classes can be used to classify the inconsistency rules. Section 4.3 then presents an example of this mechanism and highlights its benefits for building an incremental checker.

4.1 Partitioning of Operations

To reduce the set of inconsistency rules to re-check we rely on the fact that each rule is concerned by a limited set of possible construction operations. A re-check will only be necessary if at least one of these operations has been used in the model increment.

For instance, the `OwnedElement` inconsistency rule is impacted by `setReference` operations that modify the values of the `ownedElement` set of an element. As a consequence, this rule should only be re-checked if the model increment changes the values of the `ownedElement` reference set of an element. Any other operation in the model increment will not affect the evaluation result of the inconsistency rule.

In order to analyze inconsistency rules and to identify the operations that impact them, we propose a partitioning of construction operations. Given a meta-model MM and the set $\mathcal{O}_{MM}$ of all construction operations that can be performed to build model instances of this meta-model, we propose the partition $\mathcal{P}_{impact}(\mathcal{O}_{MM})$ of $\mathcal{O}_{MM}$. Two construction operations o_1 and o_2 belong to the same equivalence class if and only if : (i) o_1 and o_2 both create model element instances of the same meta-class; or (ii) o_1 and o_2 both change the values of the same reference; or (iii) o_1 and o_2 both change the values of the same property; or (iv) o_1 and o_2 both delete a model element.

This partition is finite since a meta-model holds a finite number of meta-classes and each of them holds a finite number of properties and references. The partition can be automatically computed for any meta-model based on the following guidelines:

- for each non abstract meta-class M there is an equivalence class C_M that contains all the creation operations of instances of this meta-class,
- for each property p there is an equivalence class SP_p that contains all the operations setting the property value.
- for each reference r there is an equivalence class SR_r that contains all the operations setting the reference value.
- a final equivalence class D contains all the delete actions regardless of the metaclass of the deleted model element.

The first column of figure 4 represents the partition $\mathcal{P}_{impact}(\mathcal{O}_{CMM})$ of the CMM metamodel.

4.2 Impact Matrix

The partition $\mathcal{P}_{impact}$ is used to identify the operations that may impact an inconsistency rule. From a conceptual point of view, an inconsistency rule defines a selection of specific operations within a sequence of construction operations. This selection can be abstracted by a set of equivalence classes of the partition. We name this set the corresponding equivalence classes of a rule.

For example, for inconsistency rule `OwnedElement`, the corresponding set of equivalence classes is the singleton $\{SR_{ownedElement}\}$. For inconsistency rule `OwnedParameter`, $\{SR_{ownedParameter}, SP_{direction}\}$ is the corresponding equivalence set.

We can visualize the relation between equivalence classes and inconsistency rules by a matrix where the equivalence classes represent the rows and the inconsistency rules represent the columns. The matrix contains boolean values indicating the presence of a potential impact. The impact matrix for our CMM metamodel and the two inconsistency rules is shown in Figure 4.

This impact matrix can be used as a filter on the inconsistency rules that need to be re-checked after each model increment. For each operation contained in the model increment, the corresponding equivalence class is selected and the matrix is consulted to determine which rules need be re-checked. The impact matrix ensures that all rules whose evaluation may have changed will be re-checked. It should be noted that our approach only needs to store the impact matrix to run. The size of this impact matrix depends only on the number of rules and equivalence classes.

It should also be noted that our approach is a conservative approximation. It is possible that the impact matrix identifies rules to re-check even if their evaluation is not changed by the increment. Nonetheless, our approach effectively reduces the set of rules needed to be re-checked, thereby avoiding a waste of time on performing useless computations. We will present performance results in section 5.

	ownedElement	ownedParameter
$C_{Package}$	false	false
C_{Class}	false	false
$C_{Operation}$	false	false
$C_{Parameter}$	false	false
SP_{Name}	false	false
$SP_{Direction}$	false	**true**
$SR_{ownedElement}$	**true**	false
$SR_{ownedMember}$	false	false
$SR_{ownedProperty}$	false	false
$SR_{ownedParameter}$	false	**true**
D	false	false

Fig. 4. Impact matrix for the CMM metamodel

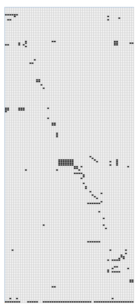

Fig. 5. Impact matrix for the UML metamodel

4.3 Example

For the sequence σ_c of Figure 3 and the inconsistency rules of section 2.1, the model is consistent. Let δ be the model increment of Figure 6 that creates two parameters and associates them with the 'send' operation through the `ownedParameter` reference. The first and second construction operations of δ belong to equivalence class $C_{Parameter}$. The third operation belongs to equivalence class $SR_{ownedParameter}$. The fourth operation belongs to equivalence class $SR_{ownedElement}$. The impact matrix informs us that the rules `ownedElement` and `ownedParameter` have to be re-checked. Performing the re-check informs us that the model remains consistent after having applied the increment.

1 create(pa1,Parameter)
2 create(pa2,Parameter)
3 setReference(o1,ownedParameter,{pa1,pa2})
4 setReference(o1,ownedElement,{pa1,pa2})

Fig. 6. a first model increment δ on σ_c

1 setProperty(pa1,direction,{'return'})
2 setProperty(pa2,direction,{'return'})

Fig. 7. a second model increment δ' on $\sigma_c.\delta$

Now, consider the second increment δ' on $\sigma_c.\delta$ in Figure 7 that changes the direction of the parameters. Both construction operations of δ' belong to equivalence class $SP_{Direction}$. The impact matrix informs us that only `ownedParameter` rule needs to be re-checked. Computing the re-check only for this rule informs us that the model is inconsistent only for `ownedParameter`.

Our approach is centered around an impact matrix that expresses relationships between inconsistency detection rules and their equivalence classes. This matrix may be generated automatically, but such a generation depends on the language that is used to define the inconsistency rules. Indeed, the more expressive the language used to express the rule is, the more complex the automatic generation of the matrix will be. We will present in section 5 how we generated the impact matrix of UML 2.1 in a semi-automated way.

5 Validation

5.1 Prototype Implementation

In [9], we presented a global model inconsistency checker that has been realized using Prolog. Inconsistency rules were translated into Prolog queries and model construction operations were translated into Prolog facts. The global inconsistency checker has been integrated into the modeling environments Eclipse EMF and Rational Software Architect. It has been written in Java and is coupled with SWI-Prolog. From any given model, a model construction operation sequence is generated and added to the fact base. The Prolog engine then executes all queries representing inconsistency rules and returns the results to the user.

The Prolog query presented below corresponds to the inconsistency rule `OwnedParameter` we introduced in Section 2.1 to identify operations that own more than one 'return' parameter:

```
ownedParameter(X) :-
  lastCreate(X,Operation),
  lastSetReference(X,ownedParameter,L),
  lastSetProperty(Y,direction,'return'),
  lastSetProperty(Z,direction,'return'), Y\=Z,
  member(Y,L), member(Z,L).
```

When evaluating this query, Prolog returns all `X` such that `lastCreate(X, Operation)` is true in the sequence. For each identified operation `X`, Prolog will evaluate whether there are any pairs `(Y,Z)` of distinct return parameters owned by the operation. If the query returns a result for `X`, then the model is inconsistent since there is at least one operation in the resulting model that owns two return parameters.

The incremental checker we propose in this paper follows the architecture of the global inconsistency checker. It is also based on Prolog, the inconsistency rules are Prolog queries and the model construction operation sequences are stored in a Prolog fact base. The incremental checker differs from the global checker by relying on the impact matrix and by working with an extensible fact base in which new facts can be added dynamically. The incremental checker receives as input a sequence of construction operations that corresponds to a model increment of a sequence that is already stored in the fact base. It parses all operations of the increment. For each of them it uses the information stored in the impact matrix to mark all inconsistency rules that require a re-check. Once all operations of the model increment have been parsed, it is added to the fact base. The user is then asked whether he wants to perform an incremental re-check or whether he prefers to continue working with a possibly partially inconsistent model.

5.2 Case Study

The UML impact matrix. We have validated our approach on the classes package of the UML 2 meta-model. The UML 2 classes package is composed of 55 meta-classes required to specify UML class diagrams. Those 55 meta-classes define a partition into 177 equivalence classes (cf. Section 4.1). The classes package defines 58 OCL constraints that we have considered as inconsistency rules. We translated these OCL constraints into Prolog queries and then built the impact matrix. The dimension of this matrix is 177×58.

In order to minimize errors when building the UML 2 matrix (which is quite big), we partially automate its construction. For that, we implemented a matrix builder that inputs inconsistency rules specified in Prolog and returns the corresponding impact matrix. It functions roughly as follows: (1) by default, all matrix values are set to `false`; (2) if the parsed inconsistency rule uses a `lastSetReference` or `lastSetProperty` construction operation, in the column corresponding to the rule in the matrix, the equivalence class of the operation is set to `true`; (3) if the rule uses a `lastCreate`, in the column corresponding to the rule in the matrix, the equivalence class of create operation as well as the equivalence class D are set to `true`.

Figure 5 presents a screenshot of the UML 2 impact matrix where the 58 rules represent the columns and the 177 equivalence classes represent the rows. For the sake of visibility, a black square represents true while a white one represents false. It should be noted that the rules and the equivalence classes are ordered according to the meta-classes defining them. For instance, in the upper left corner of the matrix appear the rules that are defined in the `Association` meta-class and the equivalence classes of corresponding construction operations (i.e., `create(Association)` and `setReference(endType)`). The last line corresponds to the `delete` equivalence class; that's why it is quite black. Moreover, as inconsistency rules defined in a meta-class often use construction operations corresponding to the meta-class, there is a kind of diagonal of `true` values in the matrix. It should be noted that the block of 4×7 `true` values in the middle of the matrix corresponds to the rules defined in the `MultiplicityElement` meta-class. This meta-class defines 7 inconsistency rules that specify the correct values of `lower` and `upper` multiplicities. Therefore, the `setProperty(lower)` and `setProperty(upper)` appear in all theses rules. Finally, one can observe that the matrix is very sparse. As we will explain in the next subsection, many inconsistency rules are impacted by only a few equivalence classes.

Analysis of the impact matrix. Figure 8 is derived from the impact matrix and shows, for each rule, the number of equivalence classes that impact each inconsistency rule. The rules are shown on the x-axis, and are ordered according to the severity of their impact: one rule is impacted by 11 equivalence classes, one by 7 equivalence classes, two rules are impacted by 6 equivalence classes, 74.1% of the rules are impacted by 3 to 5 equivalence classes, and 18.9% of the inconsistency rules are impacted by 1 or 2 equivalence classes.

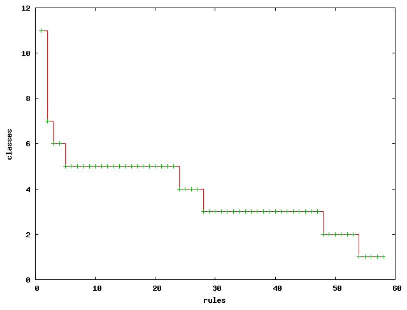

Fig. 8. Number of equivalence classes that impact a rule

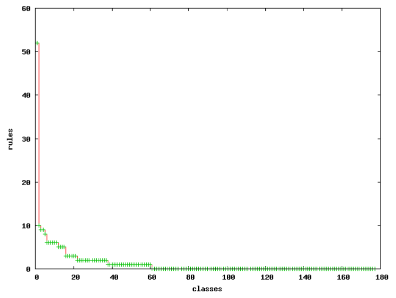

Fig. 9. Number of rules impacted by each equivalence class

Figure 9 is also derived from the impact matrix and shows, for each equivalence class, the number of rules that are impacted by it. 66.1% of the equivalence classes do not affect model consistency, 22.0% of the classes impact 1 or 2 inconsistency rules, 9.0% of the classes impact 3 to 6 rules, 2.2% impact 8 to 10 rules and only the `delete` operation impacts almost all inconsistency rules. The case of the `delete` operation is particular. It really impacts nearly all UML OCL constraints but can only be performed on model elements that are not referenced by any other model element.

Analysis of the rule complexity. Next to this static analysis of the impact matrix, we have performed a complexity analysis of the inconsistency rules. Indeed, not all rules have the same complexity. In order to measure the complexity of a rule, we used a benchmark of the time needed to check each rule for different sizes of model chunks.

It appears that 13 rules out of 58 (22,4%) take much more time than the others to be checked. For a model size around three hundred thousand model elements (about 1.9 million operations), each of those 13 rules takes more than one second to be checked; all others need only a few milliseconds. The `ownedParameter` and the `ownedElement` rules we presented in the previous section belong to those 13 time-consuming rules. A manual inspection of those 13 rules revealed that 3 have a quadratic time complexity and the others have at most a linear complexity. The `ownedElement` rule we presented in the previous section is one of the three quadratic rules. The second one specifies that classifiers generalization hierarchies must be directed and acyclical, and the third one specifies that all members of a namespace should be distinguishable within it.

Scalability analysis. We stress tested our incremental checker on a real, large-scale UML model. A huge UML class model was obtained by reverse engineering the Azureus project, which possesses a messy architecture. The model construction sequence for this UML model contained about 1.9 million model construction operations.

We performed a static analysis of the construction operation sequence of the Azureus class model. According to our impact matrix, each rule is impacted on average by 42000 operations of the construction sequence. This means that, statistically, adding a new operation will have a probability of about 3% to require re-checking an inconsistency rule.

We also executed a runtime test of our incremental checker following the test performed by Eyged [6]. This test consists of loading a complete model and simulating all possible modifications that can be performed on all the model elements. Next, for each modification, an incremental check is performed. We have performed this runtime test on our Azureus model. As the Azureus model is a huge model, there are 1809652 modifications that can be realized. Those modifications have been automatically generated and for each of them an incremental check has been performed. The same test has been repeated six times in order to filter out possible noise. The result of this runtime test is that the worst time is 50.52 seconds (almost 1 minute), the best time is less than 0.1 ms (the

Table 1. Timing results in milliseconds (averaged over 6 runs) for incrementally checking the impact of modification operations applied to Azureus

model size	number of operations	worst result	best result	average result
part 1	380866	38204	≈ 0	1722
part 2	761732	38884	≈ 0	2677
part 3	1142598	41096	≈ 0	3725
part 4	1523464	47715	≈ 0	5168
full model	1904331	50521	≈ 0	5984

time needed to look in the matrix that no rule needs to be re-checked) and the average time is 6 seconds (cf. last column of Table 1).

In order to analyze the effect of model size on the performance of our consistency checker, we have split up the Azureus model into five parts with a linearly increasing size (the fifth part corresponding to the complete Azureus model) and we have applied the same runtime test but with a set of modification operations corresponding to the size of the part. Applying the runtime test to those sub-models, it turns out that best time remains roughly the same whereas the worst time has a curious growth. In fact, we did not observe (as we would have expected) a quadratic trend that would correspond to the time needed to check the most time-consuming rules. The reason is that the inconsistencies are not uniformly distributed among parts. Our hypothesis is that the worst time depends mainly on the ordering of the operations within the sequence. Finally, we have observed that the average time increases linearly (cf. last column of Table 1). This was confirmed by a linear regression model that had a very high "goodness of fit", since the coefficient of determination $R^2 = 0.994$ was very close to 1.

Without anticipating our conclusion, those timing results seem to show that, if the inconsistency rule set contains complex rules (such as `ownedElement`), once the model size becomes important (in the order of millions operations), the time needed to perform an incremental check cannot be instantaneous and continues to increase as the model size increases.

6 Related Work

Egyed proposed a framework dedicated to instant inconsistency detection [6]. This framework monitors the model elements that need to be analyzed during the check of an inconsistency rule. If an inconsistency is detected, all the relevant model elements are inserted in a corresponding "rule scope" in order to keep track of them (a rule scope defines a relation between one inconsistency detection rule and the set of model elements that need to be analyzed to evaluate this rule). After a set of modifications, the framework traces the rule scopes that are impacted by the modifications and then automatically re-checks the corresponding rules. This allows to reduce the set of rules to re-check and the set of model elements to analyze. Egyed presents very efficient performance charts for his approach but also makes the observation that such results are due to the rules that have been considered. Indeed, all considered rules have only one root model element and their check only needs a bounded set of model elements linked either directly or indirectly to the root. With such rules, the size of the rule scope scales and the time needed to perform the check is almost instantaneous, independently of the model size. This is confirmed by our findings, but we would like to stress that not all inconsistency rules are of this kind. If we would apply Egyed's approach to the `OwnedElement` rule on our sample model (cf. figure 2), 4 "rule scopes" will be built (one per model element). The size of those rule scopes will depend on the model size (the rule scope for the `Azureus` package will own all elements of the model). Now, if we consider that a modification changes the name of the `send` operation then three rule scopes will be impacted (the ones of the `send` operation, of the `Server` class and of the `Azureus` package). Then the complex `ownedElement` rule will be checked three times and the check will require some time. To conclude, the more complex the inconsistency rules and the bigger the models, the less efficient Egyed's approach (or any other approach, for that matter), becomes.

Wagner et al. also provide a framework for incremental consistency checking in [16]. The framework monitors change events and tries to match them against detection rules that are defined as graph grammar patterns. If a match is detected then the rule is automatically re-checked. Wagner does not provide any performance analysis and does not ensure that his approach is scalable. Indeed, Wagner indicates that rules should not be time consuming in order to not block the user while he is building his models.

In [17] is presented an OCL incremental checker. The authors describe how to exploit the OCL description language to work on consistency invariant. The approach can be compared to ours because it enables to determine for each invariant the set of impacting OCL change events. And secondly, it describes how to compute an optimized invariant recheck code for each impacting change. However, OCL description language has a limited usage as it can only describe monocontextual inconsistencies, in the context of software architecture models it is advocated to target multi-context/multi-paradigm inconsistencies as presented in [18,19].

In the database community, incremental consistency checking is an important research topic that has been addressed by various authors over the years. Their main goal is to preserve data integrity, and to detect whether database updates violate integrity constraints. For example, [20] proposed a logic-based approach, implemented in Prolog, to check integrity of deductive databases. We acknowledge that there is a lot to learn from database research, even though the focus for software models is different, since inconsistencies are omnipresent and inevitable during the modeling process [3], implying that we need more flexible techniques for managing and allowing inconsistencies.

7 Conclusion

In this paper we proposed an incremental inconsistency checker that is based on a sequence of model construction operations. Our approach consists of analyzing the modifications performed on a model in order to identify a subset of inconsistency rules that need to be re-checked. The analysis is based on an impact matrix that represents dependencies between construction operation equivalence classes and inconsistency rules. Thanks to this matrix, a user can instantaneously know if the modifications he performs may or may not impact an inconsistency rule. With such knowledge he can then decide whether and when to execute the incremental check of impacted rules. Such an incremental check typically requires considerably less time than a full check. Moreover, our incremental checker scales up to huge models, as the only information required for it to run is stored in the impact matrix.

The definition of the impact matrix relies only on the meta-model and the inconsistency rules; it does not depend on the state of the models that are checked. Our incremental inconsistency checker can even consider several meta-models simultaneously in a homogenous way, since we represent models as construction operation sequences defined independently of any meta-model.

We aim to improve our incremental checker in two ways. First, we aim to classify inconsistency rules according to their severity and complexity. With such a classification, users will have more information to decide whether and when to re-check inconsistency detection rules that have been marked by our incremental checker. As a second improvement, we can not only reduce the set of rules to re-check but also the set of model elements to consider during the analysis. This would enable each rule to re-check only a relevant fragment of the whole model. Our objective is then to integrate our approach with an incremental checker such as the one proposed in [6].

The results we obtained can also contribute towards computer-supported collaborative work. Indeed, we have observed that many model construction operations are safe regarding inconsistency rules. In other words, those operations have no major negative impact on the model consistency, and can thus be performed by any one at any time. Hence, it would make sense to define a locking and transaction mechanism on top of construction operations instead of model elements in order to improve support for collaborative work.

References

1. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
2. Finkelstein, A.C.W., et al.: Inconsistency handling in multiperspective specifications. IEEE Trans. Softw. Eng. 20, 569–578 (1994)
3. Balzer, R.: Tolerating inconsistency. In: Proc. Int' Conf. Software engineering (ICSE 1991), vol. 1, pp. 158–165 (1991)
4. Fradet, P., Le Metayer, D., Peiin, M.: Consistency checking for multiple view software architectures. In: Proc. Joint Conf. ESEC/FSE 1999, vol. 41, pp. 410–428. Springer, Heidelberg (1999)
5. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: Proc. Int'l Conf. Software Engineering (ICSE 2003), Washington, DC, USA, pp. 455–464. IEEE Computer Society Press, Los Alamitos (2003)

6. Egyed, A.: Instant consistency checking for UML. In: Proceedings Int'l Conf. Software Engineering (ICSE 2006), pp. 381–390. ACM Press, New York (2006)
7. Mens, T., et al.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
8. Malgouyres, H., Motet, G.: A UML model consistency verification approach based on meta-modeling formalization. In: SAC 2006, pp. 1804–1809. ACM, New York (2006)
9. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Detecting model inconsistency through operation-based model construction. In: Robby (ed.) Proc. Int'l Conf. Software engineering (ICSE 2008), vol. 1, pp. 511–520. ACM Press, New York (2008)
10. Egyed, A.: Fixing inconsistencies in UML design models. In: Proc. Int'l Conf. Software Engineering (ICSE 2007), pp. 292–301. IEEE Computer Society, Los Alamitos (2007)
11. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. Handbook of Software Engineering and Knowledge Engineering, 329–380 (2001)
12. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logics to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
13. Elaasar, M., Brian, L.: An overview of UML consistency management. Technical Report SCE-04-18 (August 2004)
14. OMG: Unified Modeling Language: Super Structure version 2.1 (January 2006)
15. OMG: Meta Object Facility (MOF) 2.0 Core Specification (January 2006)
16. Wagner, R., Giese, H., Nickel, U.A.: A plug-in for flexible and incremental consistency management. In: Workshop on consistency problems in UML-based Software Development - Satellite Workshop of MODELS (2003)
17. Cabot, J., Teniente, E.: Incremental evaluation of ocl constraints. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 81–95. Springer, Heidelberg (2006)
18. ISO/IEC 42010: Systems and software engineering architectural description. ISO/IEC WD3 42010 and IEEE P42010/D3 (2008)
19. Boiten, E., et al.: Issues in multiparadigm viewpoint specification. In: Foundations of Software Engineering, pp. 162–166 (1996)
20. Kowalski, R.A., Sadri, F., Soper, P.: Integrity checking in deductive databases. In: Proc. Int'l Conf. Very Large Data Bases (VLDB), pp. 61–69. Morgan Kaufmann, San Francisco (1987)