Towards Automated Inconsistency Handling in Design Models

Marcos Aurélio Almeida da Silva[1,⋆], Alix Mougenot[1], Xavier Blanc[1], and Reda Bendraou[1]

LIP6, UPMC Paris Universitas, France

Abstract. The increasing adoption of MDE (Model Driven Engineering) favored the use of large models of different types. It turns out that when the modeled system gets larger, simply computing a list of inconsistencies (as provided by existing techniques for inconsistency handling) gets less and less effective when it comes to actually fixing them. In fact, the inconsistency handling task (i.e. deciding what needs to be done in order to restore consistency) remains largely manual. This work is a step towards its automatization. We propose a method for the generation of *repair plans* for an inconsistent model. In our approach, the depth of the explored search space is configurable in order to cope with the underlying combinatorial characteristic of this problem and to avoid overwhelming the designer with large plans that can not be fully checked before being applied.

1 Introduction

As an effect of the increasing adoption of MDE (Model Driven Engineering), large-scale industrial projects are currently being developed by hundreds of people and make use of several models instance of different meta-models (e.g. SysML, UML, Petri nets, architecture, work, business process) [1]. In such a context, it turns out that inconsistencies that may exist in models have been revealed as one of the main development problems [2] and that is why developing techniques for inconsistency management becomes so important.

A model is considered to be inconsistent if and only if it contains undesirable patterns, which are specified by the so called inconsistency rules [3]. Even if there are several variants of inconsistency rules such as well-formedness rules of [4], structural rules of [5], detection rules of [6], syntactic rules of [7], and inconsistency detection rules of [8], approaches that deal with detection of inconsistencies irremediably consists in browsing the model in order to detect undesirable patterns. However, as defined by [4], inconsistency management not only consists in the detection of inconsistencies but also in their handling. Indeed, once inconsistencies have been detected on models, they have to be resolved.

The work we present in this paper focuses on the *handling of inconsistencies*, which consists in automating the modification of a model in order to make

⋆ This work was partly funded by ANR project MOVIDA Convention N° 2008 SEGI 011.

B. Pernici (Ed.): CAiSE 2010, LNCS 6051, pp. 348–362, 2010.

it consistent again. The first challenge we address, in order to make a model consistent, is to identify the potential changes that can be applied to fix detected inconsistencies. Regarding this challenge, Nentwich has clearly shown that, for any given set of inconsistencies, there exists an infinite number of possible ways of fixing it [9]. Therefore for efficiency reasons, this challenge is more related to narrowing the scope of the identification rather than to enumerate all possible resolutions.

The second challenge we address is to measure the impact of a given identified change. Indeed, as Mens has shown in [6], a change that fixes one inconsistency may introduce new ones and therefore will be counter productive. This second challenge consists then in filtering out non productive changes in order to keep only productive ones.

The last challenge we address consists in computing the execution order of a set of productive changes that do solve a set of inconsistencies and that do not introduce new ones. This last challenge consists in choosing an appropriate order of execution of the potential productive changes and is therefore a combinatorial challenge. To sum up, we argue that the *handling of inconsistencies* aims at providing what we call *repair plans*, which are sequences of concrete changes to be performed over a given model and that fix existing inconsistencies without introducing new ones.

In this paper we propose an approach for automatic generation of *repair plans*. Our main motivation is to assist the developers while they build their models. We argue that in such context being able to generate partial short plans very quickly is a desirable feature.

Our proposal is based on Praxis[8], our model inconsistency detection approach. In Praxis, the model is represented as the sequence of actions executed by the user in order to build it. The *repair plans* we propose to generate are then sequences of Praxis actions. The key contribution of our work is leveraging a search algorithm that is optimized to find the best plan (the shortest plan that fixes the bigger number of inconsistencies) by exploring a limited and configurable subset of all possible plans. Our approach has been prototyped into the Praxis environment that runs on top of the Eclipse EMF plataform. It proposes *repair plans* to UML developers while they build their UML models.

This paper is organized as follows: in Section 2, we present a motivating example of the inconsistency handling task. In Section 3, we present the Praxis formalism. In Section 4, we present our approach for automated inconsistency handling. In Section 5, we present our prototype implementation of this approach and the runtime results in applying these techniques in a series of randomly generated UML2 models. In Section 7, we conclude this paper after the section 6 that presents the related works.

2 Motivating Example

This section illustrates the problem of inconsistency handling in an example. All models in this paper are instances of the subset of the UML2 meta-model[10] displayed in Figure 1.

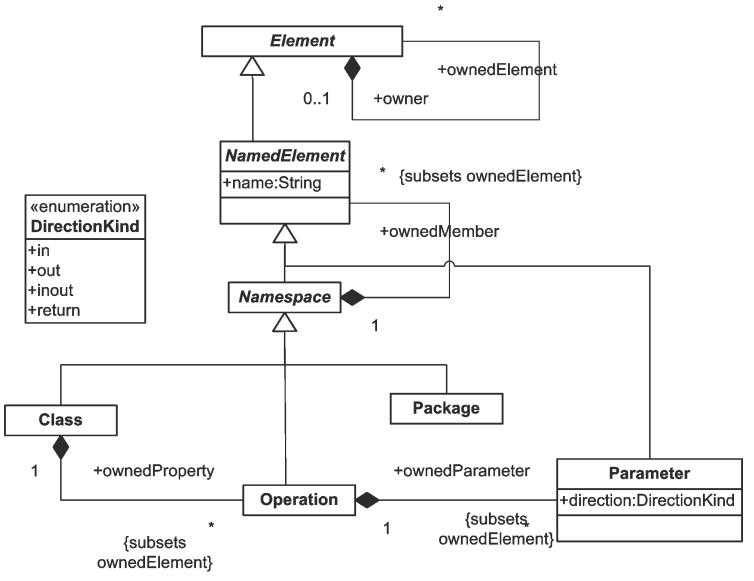

Fig. 1. A simplified fragment of the UML2 meta-model

The root concept of UML21 is the one of *model element* that is represented by the meta-class *Element*. Everything in a UML2 model is an instance of *Element*. An *Element* may contain a set of other elements. The *NamedElement* meta-class represents the elements that have a name (stored in the string attribute *name*). A *Namespace* is a named model element that serves as a space of names for the named elements it contains. Three classes inherit from *Namespace*: *Package*, *Class* and *Operation*. A *Package* is a container for other elements. A *Class* represents the object-oriented concept of a class of objects and serves as a namespace for its operations. An *Operation* represents an operation that is provided by the instances of a class and represents a namespace for its parameters. Every *Parameter* has a direction of one of four *kinds*: *in*, *out*, *inout* and *return*.

Figure 2 presents a UML model used as an example in this paper. This model contains a package called *Azureus* that owns two classes: *Client* and *Server*. The *Server* class defines an operation called *send*(). Figure 3 presents this model after some changes have been done on it in order to make it inconsistent. Those changes break the following well-formedness rule defined in the UML2 specification: *"All the members of a Namespace are distinguishable within it"*.

In the changed model, a new class named *Client* has been added. This creates an inconsistency since there is now two classes with the same name. In the rest of this paper, this first inconsistency is named `namespace-1`. In the changed model, a new operation named *send*() has been added in the class *Server*. This creates another inconsistency since there is now two operations with the same name. In the rest of this paper, this second inconsistency is named `namespace-2`.

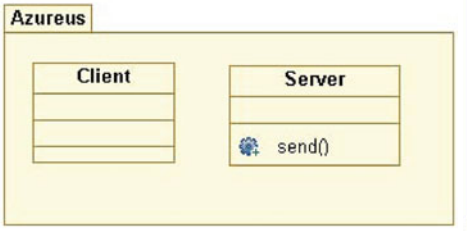

Fig. 2. Sample UML2 model

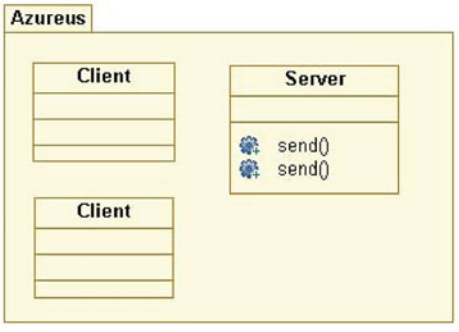

Fig. 3. Sample modified UML2 model

The first step in handling these inconsistencies is enumerating the repair changes that could solve the detected inconsistencies (i.e. deciding *how* to solve them). In this example, at least four sample repair changes[1] can be enumerated:

1. Rename one of the *Client* classes to *class*1.
2. Rename one of the *Client* classes to *Server*.
3. Rename one of the *send*() operations to *operation*1().
4. Rename one of the *send*() operations to *operation*2().

Each of the four repair changes fix some particular inconsistency by renaming one of the offending elements: repair changes 1 and 2 fix `namespace-1` and repair changes 3 and 4 fix `namespace-2`. The next step is thus to analyze *what* are the impacts of each repair change. Notice that the repair change 2 introduces a new inconsistency of the same kind by making the old class *Client* indistinguishable with the the old class *Server* (let us name this inconsistency `namespace-3`).

Finally, a analyze has to be done in order to decide *when* each of the repair change has to be executed, i.e. which change will be executed to fix each inconsistency and in which order. For example, if the repair changes 2 and 4 are executed in this order, this will solve both original inconsistencies, but will lead to a new one: `namespace-3`. This analyze will result that, for this sample, four possible repair plans can be executed: $1-3$, $1-4$, $3-1$, and $4-1$.

3 The Praxis Formalism

As described in [8], in the Praxis formalism, models are represented as sequences of elementary actions needed to construct each model element. Each action is

[1] Notice that these are just four among the infinite number of repair changes that actually solve one of the inconsistencies in the model. For instance, for every string *s* different from *Client* the plan *"Rename one of the Client classes to s."* is a valid one.

```
create(p1,package,1)
addProperty(p1,name, 'Azureus',2)
create(c1,class,3)
addProperty(c1,name, 'Client',4))
create(c2,class,5)
addProperty(c2,name,'Server',6)
addReference(p1,ownedMember,c1,7)
addReference(p1,ownedMember,c2,8)
addReference(p1,ownedElement,c1,9)
addReference(p1,ownedElement,c2,10)
create(o1,operation,11)
addProperty(o1,name, 'send', 12)
addReference(c2, ownedProperty, o1, 13)
addReference(c2, ownedElement, o1, 14)
```

Fig. 4. Model construction operation sequence

annotated with a timestamp which indicates the moment when it was executed by the user. The actions are inspired by the MOF reflective API [11].

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 timestamp t. The $addProperty(me, p, value, t)$ and $remProperty(me, p, value, t)$ add or remove the value $value$ to or from the property p of the model element me at timestamp t. Similarly, the actions $addReference(me, r, target, t)$ and $remReference(me, r, target, t)$ add or remove the model element $target$ to or from the reference r of the model element me at timestamp t.

Figure 4 presents the sequence of basic actions that construct the model presented in Figure 2. The action at timestamp 1 creates the package $p1$. The second action sets its name to $Azureus$. The actions with the timestamps 3 and 4 and those withe the timestamps 5 and 6 create the classes $c1$ and $c2$ and set their names to $Client$ and $Server$ respectively. The actions with the timestamps 7 and 10 state that both classes are owned by the package $p1$. The actions with the timestamps 10 and 11 create the operation $o1$ whose name is $send$ and the actions with timestamp 13 and 14 add it to the class $c2$.

Notice that this sequence is not unique, in the sense that there are usually many sequences that construct the same model. For example, changing the places of the actions with timestamps 7 and 10 and the actions with the timestamps 11 and 14 would result in the same model.

3.1 Inconsistency Detection in Praxis

In Praxis, inconsistencies are detected by the means of logic rules over the sequence of model construction actions that identify undesired patterns in it. The actions in the sequence are referenced by the means of logical predicates (expressed in Prolog in our prototype) that match them. For example, the Prolog

```
namespaceOCL1(ME1, ME2) :-
        lastAddReference(NS,ownedmember,ME1),
        lastAddReference(NS,ownedmember,ME2),
        ME1 \== ME2,
        not(distinguishable(ME2,ME1)).
```

Fig. 5. Inconsistency detection rule in Praxis

query `create(X, package, 1)` would result in the answer `X=p1` in the model represented in Figure 4.

As syntactic shortcuts, the '`last`' prefix denotes actions that are not followed by other actions canceling their effects. Moreover, for each predicate referring to a Praxis operation, there is a similar one without the timespamp parameter. For instance, a `lastCreate(me, class)` operation is defined as a `create(me, class,t)` operation that is not followed by a `delete(me,t)` operation; and a `lastAddProperty(me, name, val)` operation is defined as a `addProperty(me, name, val,t)` operation for which the value of the *name* property of `me` in the model corresponds to `val`.

Figure 5 displays an example of inconsistency detection rule. This rule defines the predicate `namespaceOCL1(ME1, ME2)` that detects pairs of model elements `ME1` and `ME2` that are owned by the same namespace `NS` but are not distinguishable. The rules defining the predicate `distinguishable(ME1, ME2)` were omitted for the sake of brevity. Let us just consider that it holds if and only if `ME1` and `ME2` are instances of different meta-classes or if they are instances the same meta-classes but have different names.

4 An Approach for Inconsistency Handling in Praxis

In Praxis, it is worth to assume that model inconsistencies are *introduced* by user's actions that violate some of its constraints or by *not executing* actions required by these constraints.

In order to generate a repair plan to fix these inconsistencies (either by undoing undesired actions or by adding omitted ones), our approach answers the three following questions (i) how to detect the actions that caused inconsistencies? (ii) how to enumerate the possible ways of changing a given inconsistent action? and (iii) how to generate a repair plan for the model sequence from the list of possible ways of changing the model?

4.1 How to Detect the Actions That Caused Inconsistencies?

Our approach to this problem is to adapt the inconsistency rules presented in Section 3.1 so that, instead of pointing out the problematic *elements* in the model they are going to identify the problematic *actions* which have (probably) *caused* the problem.This is done by the means of the *cause detection rules*.

```
namespaceOCL1(Cause) :-
        lastAddReference(NS,ownedmember,ME1, TS1),
        lastAddReference(NS,ownedmember,ME2, TS2),
        lastAddProperty(ME1, name, NM1, TS3),
        lastAddProperty(ME2, name, NM2, TS4),
        ME1 \== ME2,
        not(distinguishable(ME2,ME1)),

        Causes = [ addReference(NS,ownedmember,ME1, TS1),
                   addReference(NS,ownedmember,ME2, TS2),
                   addProperty(ME1, name, NM1, TS3),
                   addProperty(ME2, name, NM2, TS4)       ],
        member(Cause, Causes).
```

Fig. 6. Cause detection rule in Praxis

Figure 6 presents the cause detection rule written from the inconsistency detection rule initially displayed in Figure 5. This new rule identifies four possible causes for the detected inconsistencies: the actions that added the model elements `ME1` or `ME2` to the same namespace `NS` or the actions that defined their names. Notice that, for this rule, the version of the Praxis predicates with explicit timestamps has been used because there is a need to identify unambiguously the action that caused the inconsistency.

In most cases a trivial *cause detection rule* can be generated from an inconsistency detection rule: just point all actions used to prove an inconsistency as possible causes of it. This is a possible strategy, although suboptimal, since not all involved actions are necessarily responsible for an incoherence. For example, let us take the following well-formedness rule from the UML2 metamodel: *"The visibility of all features owned by an interface must be public."*

This rule is specified as the following cause detection rule:

```
interfaceOCL1(Cause) :-
        lastCreate(X,interface),
        lastAddReference(X,feature, F),
        lastAddProperty(F,visibility,V,TS),
        not(V='public'),
        Cause = addProperty(F,visibility,V, TS).
```

This rule states that if `X` is an interface and `F` is one of its features and its visibility is not `public` then the cause of this inconsistency is the action that defines its visibility. For each inconsistency of such kind that is proved, three actions need to be inspected: a *create*, an $addReference$ and an $addProperty$; although only one of them is actually responsible for the inconsistency. Therefore, our manually written rule would prune two thirds of the search space, by avoiding looking for repair plans that fix actions that did not caused the problem.

4.2 How to Enumerate the Possible Ways of Changing a Given Inconsistent Action?

We propose to use generator functions that determine a set of lists of actions that cancel the effects of a given inconsistency causing action in the model. These manually written rules are mostly independent of the cause detection rules, since they do not have to consider impact of the changes they propose, neither to other inconsistencies that might exist in the model nor to the set of cause detection rules.

Let us analyze the following partial definition of the generator function:

```
generate(addProperty(E, name, OldName, TS),
             [remProperty(E,name, OldName),
              addProperty(E, name, NewName)]) :-
                        lastCreate(E, C),
                        randomNameGenerator(C, NewName).

generate(addProperty(E, visibility, 'public', TS),
             [remProperty(E, visibility, 'public'),
              addProperty(E, visibility, 'private')]).

generate(addProperty(E, visibility, OldVisibility, TS),
             [remProperty(E, visibility, OldVisibility),
              addProperty(E, visibility, 'public')])
              :- not(OldVisibility='public').
```

The first rule cancels inconsistencies in a `addProperty` action to the `name` field of a model element `E`. It says that every time a `addProperty(E, name, OldName, TS)` action is a source of inconsistency there is a simple plan that may fix it: removing the old value of the property by executing the action `remProperty(E,name, OldName)` and setting its name to another value `NewName` by the means of the action `addProperty(E, name, NewName)`.

Observe that this new name is generated using a random name generator accessible by the predicate `randomNameGenerator(C, Name)` such that it generates a name `Name` taking the meta-class `C` as a reference (e.g. generating a `metaclassX` name for a metaclass named `metaclass`).

The second and third rules fix inconsistencies in the `visibility` property of a model element `E`: if it was `public` the second rule suggests changing it to `private`, otherwise the third rule changes it to `public`.

As it is shown in [12], brute force generation of choices is not scalable, therefore, a well-written generator function needs to be custom-tailored in order to reduce the number of proposed choices while not introducing obvious inconsistencies (i.e. removing all values of a non-optional property).

4.3 How to Generate a Repair Plan for the Model Sequence?

The Iterative Deepening Depth-first Search Strategy (IDDFSS) [13] is a depth-first tree search algorithm that allows exploring the tree of possible repair plans

by taking into consideration the first suggested change for the first cause of inconsistency and going thus deeper and deeper in the search three until either a consistent model is found or until there is no possible action and thus backtracking.

The case when no action is possible happens when the algorithm reaches the maximum allowed depth in the search tree (i.e. when a previously defined maximum number n of execution steps has been reached, this is called a *depth-limited search*). This strategy is therefore capable of finding any repair plans that can be constructed after executing at most n steps. Indeed, if no complete repair plan exists in this limited search space, *partial* plans may also be constructed by recording the *best* (e.g. the one that generates a final model with less inconsistencies) plan found during the search.

In IDFSS, the maximum number of steps n is iteratively incremented, e.g. if the max number of steps is defined to be m, then a depth-limited search will be executed for every n between 1 and m until a repair plan that fixes all inconsistencies is found. This makes sure that the final plan was obtained with the least number of steps as possible.

On top of IDFSS we propose to adopt an heuristics that decides the next action to be fixed by ordering the inconsistency causes from the most recent to the least recent one. This design decision is based on the assumption that the user tries to maintain the model as consistent as possible at most of the time. Taken into account that inconsistencies are *introduced* in the model by actions executed later in the design process, it is reasonable to assume that the actions that were executed more recently are more likely to have introduced new inconsistencies that were not there before.

Finally, if there exists a repair plan that can be constructed with less than m execution steps the IDDFSS algorithm is guaranteed to find it. The remaining problem is thus determining the optimal value of m that is going to find a complete repair plan.

In fact, this problem is simpler then it seems at first sight since defining a precise value for m is *not necessary*. The value of m just needs to be set to be *big enough* to allow us to find a solution. Choosing a value that is *bigger* than needed has no impact on the actual execution time, since the depth-limited search will be repeated for all values from 1 to m until a complete solution is found, i.e. if a complete repair plan is found for some $n = k$ before reaching m the rest of the possible depths will not be tested.

4.4 Running Example

This section highlights our approach on the sample model displayed in Figure 3. Let us suppose that its model sequence is composed by the sequence displayed in Figure 4 appended with the following sequence:

```
create(c3, class, 15)
addReference(p1, ownedMember, c3, 16)
addReference(p1, ownedElement, c3, 17)
addProperty(c3, name, 'Client', 18)
```

create(o2, operation, 19)
addReference(c2, ownedProperty, o2, 20)
addReference(c2, ownedElement, o2, 21)
addProperty(o2, name, 'send', 22)

This sequence defines an inconsistent model, because `c2` and `c3` and `o1` and `o2` and indistinguishable in their respective namespaces. Let us detail the execution of one iteration of depth-limited search in the IDFSS algorithm in which the depth of the search tree is limited to 2.

Step 1
At this point, the cause detection rules are used to compute the list of causes of inconsistencies in the model. The list is organized in the inverse order of timestamps:

```
addProperty(o2, name, send, 22)
addReference(c2, ownedmember, o2, 21)
addProperty(c3, name, 'Client', 18)
addReference(p1, ownedmember, c3, 17)
addReference(c2, ownedelement, o1, 14)
addProperty(o1, name, send, 12)
addReference(p1, ownedmember, c1, 7)
addProperty(c1, name, 'Client', 4)
```

The search tree is going to be explored in a depth-first search, this means that each step fixes the possible causes of inconsistencies from the most recent to the older one. If no consistent model was found after exploring one possibility, the next cause should be tried. At this point, the first cause of inconsistency (`addProperty(o2, name, send, 22)`) is taken. The generator function is then used to obtain a list of possible actions to fix it.

In this case, the generator function returns only one possibility composed of two actions:

```
remProperty(o2, name, send, 23)
addProperty(o2, name, operation1, 24)
```

Those actions are then appended to our model sequence.

Step 2
Like in the step 1, this step starts by computing the new ordered list of inconsistencies. The following list is then computed:

```
addProperty(c3, name, 'Client', 18)
addReference(p1, ownedmember, c3, 17)
addReference(p1, ownedmember, c1, 7)
addProperty(c1, name, 'Client', 4)
```

Then, this process process is repeated by using the generator function in order to fix the first cause of inconsistency (`addProperty(c3, name, 'Client', 18)`) and getting the following list of actions:

```
remProperty(c3, name, 'Client', 25)
addProperty(c3, name, class1, 26)
```

At this point there are no more inconsistencies left on the model sequence so, the final repair plan is:

```
remProperty(o2, name, send, 23)
addProperty(o2, name, operation1, 24)
remProperty(c3, name, 'Client', 25)
addProperty(c3, name, class1, 26)
```

Two depth levels were explored in the search tree and it was enough to find a repair plan that fixed all inconsistencies in the model. If at the end of this limited exploration no solution is found, the execution would restart from scratch, but would explore a larger search space (e.g. a search limited to depth 3) and so on.

5 Prototype Implementation

In [8], we present the Praxis prototype. It is composed of two components: the *Sequence Builder* (which integrates to Eclipse EMF Framework and builds the model sequence from the actions executed by the user while creating a model) and the *Check Engine* (which is responsible for detecting inconsistencies).

This prototype has been extended in order to support the generation of repair plans. In particular, the *Model Fixing Agent* component has been integrated within Praxis. This component is an intelligent agent that proposes real time repair plans in order to fix the inconsistencies found in the model. The core functionalities of this component is entirely implemented in a set of Prolog rules that are packaged into an Eclipse Plug-in that interfaces with the existing Praxis plug-ins.

Figure 7 displays a screenshot of this integration. In (1) we show the class diagram presented in Figure 3 drawn using the Papyrus UML Tool integrated to Praxis. While the user is building the model, the Sequence Builder component in Praxis builds the sequence of actions. The Model Fixing Agent then watches this sequence and checks for inconsistent actions in it regularly. In (2) we see that it displays the list of inconsistencies found in the current model.

After detecting and showing the list of inconsistent actions, the Model Fixing Agent computes a repair plan for them. In (3) the four actions needed to repair the current model are listed in the order they need to be executed.

5.1 Case Study

Our approach has been stress tested with models that have been automatically generated by a mathematically grounded random model sampler [14]. These tests were executed with models of different sizes (varying from 20 to 10,000 model elements), and using different depths in the explored search space (1, 5, 10 and 15 levels) to show the impact of this parameter on the plan generation time

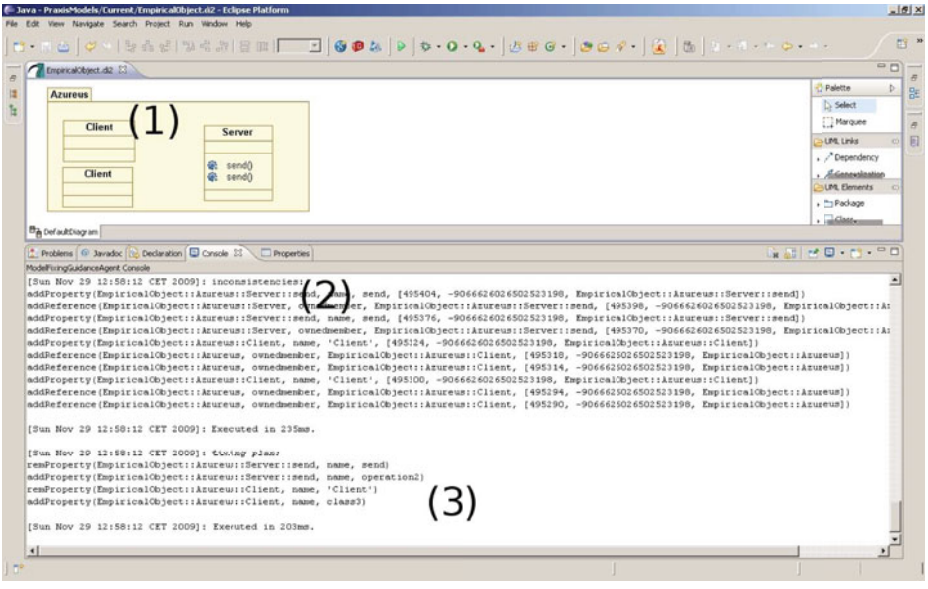

Fig. 7. Screenshot of integration with Praxis

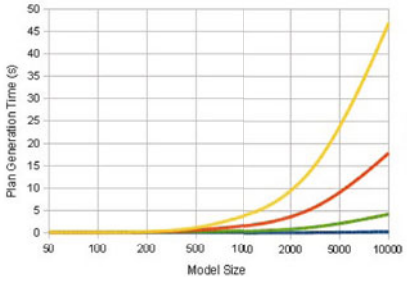

Fig. 8. Timing results in seconds

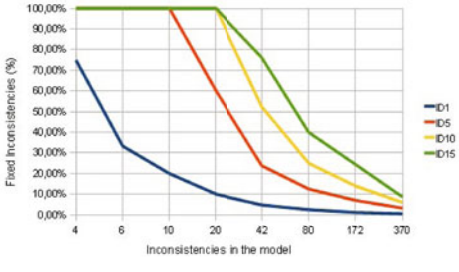

Fig. 9. Fixed inconsistencies results

and in the number of fixed inconsistencies. We manually implemented a subset of 4 of the UML2 well-formedness rules as cause detection rules and a set of 7 generator functions. Each test was executed 10 times and the average time was recorded.

Figure 8 shows the timing results of our tests. The graph clearly shows the exponential characteristic of our problem. Notice that the ID1 line (the results for exploring just one level in the search tree) is equivalent to current existing inconsistency detection approaches that are only capable of suggesting fixes for one inconsistency at a time.

Figure 9 shows the relative number of inconsistencies that is solved by our approach for each model for each maximum level of exploration. This graph

shows that the deeper is the level of the exploration the bigger is the proportion of solved inconsistencies. It also shows that for each level of exploration, there is a number of inconsistencies, such that, beyond that, only partial repair plans can be found.

6 Related Work

In [12], an approach for fixing inconsistencies in UML models is presented. In short, this approach uses a model profiler that monitors the parts of the model that were touched when consistency rules are evaluated. Those parts are then considered to be natural candidate for fixing actions when the rules are evaluated to be inconsistent. For each inconsistency, the approach explores all the possibilities of changing the monitored model parts in order to make the model consistent. The identification of possible changes are guided by the *choice generation functions*. The choices that turn the model into a consistent state are then presented to the user. Instead of trying to automatically identify the causes of the inconsistencies, our approach asks for their definition thanks to *Cause detection rule*. This definition may be automatically generated from existing inconsistency detecting rules, however, from our experience, it seems that is needs to be manually optimized afterwards. The choice generation functions of Egyed's approach inspired our generator functions. However, instead of generating just options for changing the model graph, our generator functions deliver alternative fixing plans for given actions in the model sequence.

In [9], Nentwich et al describe a framework for repairing inconsistent documents in a distributed setting. Their approach consists of defining a mapping from the logical language used to describe the inconsistency rules into a set of *repair actions* that, after being executed, will make the model consistent again. In [6], Mens et al present an approach for inconsistency management on top of graph transformation tool AGG. They detect the inconsistencies in the model by the means of the inconsistency detection rules (that tags model elements as *conflicted*) and fix inconsistencies by the means of the *resolution rules* (for each possible resolution of every kind of inconsistency there is one rule that describe how should be the model after fixed). They use then a critical pair analysis algorithm to infer dependencies between rules and aid the user in the task of fixing the model. Both approaches automate the process of defining *how* to deal with the inconsistency by proposing a set of actions that fix each of them, and they automate the definition of *what* are the impacts of the suggested repairs: they detect inconsistencies among different plans and thus discourage their application at the same time. However, the decision on the *order* of the execution of the proposed repair actions (or plans) is left to the user. In our proposal, we cope with this problem by exploring the set of generated choices and actually delivering a plan of execution of the proposed actions.

It is still noticeable that, as pointed out by [12], the use of choice functions reduces significantly the programming effort when compared to the resolution rules approach used in [6]. In that case, the number of "repair rules" is bounded

by $O(\#R * \#L_T)$ (where $\#R$ is the number of consistency rules and $\#L_T$ is the number of location types) where in our approach it reduces to $O(\#L_T)$. Notice also that the generator functions are *manually* custom-tailored to the syntactical constraints of the particular meta-model. This avoids the problem of non-scalability of approaches that compute all possible choices (such as [9] and [15]) as pointed out in [12].

7 Conclusion

In this paper we propose an approach for obtaining automatic generated *repair plans* for a given inconsistent model. Our approach is base on three main mechanisms, which are the *Cause detection rules*, the *Generator functions* and the *search algorithm*. *Cause detection rules* are used to identify actions that make the model inconsistent. Once those actions are identified, they can be repaired. It should be noted that *Cause detection rules* are directly derived from inconsistency detection rules employed in Praxis and can substitute them. Even if, for the sake of efficiency, they currently have to be manually written, they should be automatically generated from detection rules. This automatical generation is however left as as future work. *Generator functions* are used to dive the generation of repair actions and are therefore loosely coupled with the *Cause detection rules*. The *search algorithm* is used to efficiently generate repair plans, which are composed of sequence of repair action. It is fitted to build repair plans that start by fixing the most recent causes of inconsistencies. This algorithm is also parameterizable in such a way that the size of the search space to be explored during its execution can be decided beforehand.

Thanks to these three mechanisms, the modeler can, besides obtaining repair plans that correct all inconsistencies in his model, get partial plans that start by proposing fixes to the inconsistencies that were more recently introduced in the model. We argue that this capacity helps the developer when correcting models that have too much inconsistency and which would therefore require too much time to compute a complete repair plan.

Our approach is integrated into the existing Praxis environment on top of Eclipse EMF and thus is accessible to modelers using any compatible EMF based Eclipse UML Editor (such as Papyrus, that was used in this study). We currently elaborating a empirical study in order to measure the effect of our approach for industrial developers who works on UML models.

References

1. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
2. Hessellund, A., Czarnecki, K., Wasowski, A.: Guided development with multiple domain-specific languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 46–60. Springer, Heidelberg (2007)
3. Balzer, R.: Tolerating inconsistency. In: Proc. Int' Conf. Software engineering (ICSE 1991), vol. 1, pp. 158–165 (1991)

4. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: Handbook of Software Engineering and Knowledge Engineering, pp. 329–380. World Scientific, Singapore
5. 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)
6. 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)
7. Elaasar, M., Brian, L.: An overview of UML consistency management. Technical Report SCE-04-18 (August 2004)
8. 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, New York (2008)
9. 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, Los Alamitos (2003)
10. OMG: Unified Modeling Language: Super Structure version 2.1 (January 2006)
11. OMG: Meta Object Facility (MOF) 2.0 Core Specification (January 2006)
12. Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in UML design models. In: Proc. ACM/IEEE Int'l Conf. Automated Software Engineering (ASE 2008), pp. 99–108. ACM, New York (2008)
13. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education, London (2003)
14. Mougenot, A., Darrasse, A., Blanc, X.: Uniform random generation of huge metamodel instances. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 130–145. Springer, Heidelberg (2009)
15. Dam, K.H., Winikoff, M.: Generation of repair plans for change propagation. In: Luck, M., Padgham, L. (eds.) AOSE 2007. LNCS, vol. 4951, pp. 132–146. Springer, Heidelberg (2008)