Comparing Automated Planning Approaches for Model Inconsistency Resolution

Jorge Pinna Puissant[a,*], Ragnhild Van Der Straeten[a,b], Tom Mens[a]

[a] *University of Mons, 20 Place du Parc, 7000 Mons, Belgium*
[b] *Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium*

Abstract

A wide variety of approaches has been proposed to detect and resolve software model inconsistencies. In this article, we present a new type of approach that uses the artificial intelligence technique of automated planning for the purpose of resolving software model inconsistencies. We objectively compare two different variants of automated planning, progression and regression planning, and we discuss how to improve the proposed techniques further.

Keywords: automated planning, software modeling, inconsistency management, model evolution, comparison

1. Introduction

One of the main challenges in *model-driven software engineering (MDE)* is how to deal with evolving models, and how to provide more automated mechanisms to support this evolution [1]. A particular point of attention is how to manage inconsistencies in software models [2]. Such model inconsistencies are inevitable, because a (software) system description is composed of a wide variety of diverse models, some of which are developed and maintained in parallel. Our research does not focus on the activity of model inconsistency *detection*, that is becoming well-established. Instead, we address the problem of model inconsistency *resolution*. In particular, we focus on more automated ways to resolve a selection of previously identified model inconsistencies through the generation of so-called *resolution plans*.

To do this, we propose to use the technique of *Automated Planning* coming from the Artificial Intelligence domain. This technique allows the generation of possible resolution plans without the need of manually writing resolution rules. In this article, we evaluate two different planning approaches, progression planning and regression planning. We perform an objective comparison of both

[*] Corresponding author: Avenue du champ de Mars 6, 7000 Mons, +32 65 37 33 21
Email addresses: `jorge.pinnapuissant@umons.ac.be` (Jorge Pinna Puissant), `rvdstrae@vub.ac.be` (Ragnhild Van Der Straeten), `tom.mens@umons.ac.be` (Tom Mens)

planning variants to determine which approach is most promising, and we discuss on how to further advance the proposed technique and the field of model inconsistency resolution.

2. Related work

Several state-of-the-art approaches on inconsistency resolution exist. In our previous work [3] we specified resolution rules manually, which is an error-prone process. Automatic generation of inconsistency resolution actions aim to resolve this problem. Nentwich et al. [4] achieve this by generating resolution actions automatically from the inconsistency detection rules. The execution of these rules, however, only resolves one inconsistency at a time. As recognised by the authors, this causes problems when inconsistencies and their resolution are interdependent. In [5] we proposed a formal approach based on graph transformation to analyse these interdependencies.

Xiong et al. [6] define a language to specify inconsistency rules and the possibilities to resolve the inconsistencies. This requires inconsistency rules to be annotated with resolution information. Almeida da Silva et al. [7] propose an approach to generate resolution plans for inconsistent models, by extending inconsistency detection rules with information about the causes of the inconsistency, and by using manually written functions that generate resolution actions. In both approaches inconsistency detection rules are polluted with resolution information.

Instead of explicitly defining or generating resolution rules, a set of models satisfying a set of consistency rules can be generated and presented to the user. Egyed et al. [8] define such an approach for resolving inconsistencies in UML models. Given an inconsistency and using choice generation functions, possible resolution choices, i.e., possible consistent models, are generated. The choice generation functions depend on the modelling language, i.e., they take into account the syntax of the modelling language, but they only consider the impact of one consistency rule at a time. Furthermore these choice generation functions need to be implemented manually.

3. Automated Planning

In state-of-the-art approaches on inconsistency resolution, resolution rules or resolution generators need to be implemented manually and only one inconsistency at a time is considered [3, 4, 5, 6, 7, 8]. Our aim is to tackle the problem of inconsistency resolution by generating possible resolution plans without the need of manually writing resolution rules or writing any procedures that generate choices. The approach needs to generate valid models with respect to the modelling language and needs to enable the resolution of multiple inconsistencies at once and to perform the resolution in a reasonable time. In addition, the approach needs to be generic, i.e., it needs to be easy to apply it to different modelling languages. We explore *Automated Planning*, a technique coming from artificial intelligence, for this purpose [9].

Automated planning aims to create *plans*, i.e., sequences of primitive actions that lead from an initial state to a state meeting a specific predefined goal. To accomplish this, the planner decomposes the world into logical conditions and represents a state as a conjunction of literals. As input the planner needs a *planning environment*, composed of an initial state, a desired goal and a set of primitive actions that can be performed. The initial state represents the current state of the world. The goal is a partially specified state that describes the world that we would like to obtain. The actions express how each element of a state can be changed. The actions are composed of a *precondition* and an *effect*. The effect of an action is executed if and only if the precondition is satisfied. In general a planning approach consists of a representation language used to describe the problem and an algorithm representing the mechanism to solve the problem.

One way to solve planning problems consists in translating them into a satisfiability problem and using a model checker [10]. A more direct approach consists in generating a search space and looking for a solution in this space. Depending on how the state space is traversed, we can distinguish between *progression* planning and *regression* planning. *Progression planning* performs a *forward search* that starts in the initial state and tries to find a sequence of actions that reaches a goal state. *Regression planning* starts in the goal state and searches backwards to find a sequence of actions that reach the initial state. In this article, we will compare both approaches for the purpose of finding a model inconsistency resolution plan.

4. Planning for Inconsistency Resolution

There is a wide variety of modeling languages, domain-independent as well as domain-specific. As a consequence, there are many different types of, often interrelated, models that can suffer from many kinds of inconsistencies, such as structural and behavioural inconsistencies.

For our running example, we focus on one type of model only, namely class diagrams, and we focus on structural model inconsistencies. Figure 1 illustrates a simple class diagram containing two structural inconsistency occurrences of type "Inherited Cyclic Composition" (ICC) and two occurrences of type "Cyclic Inheritance" (CI) [11]. An ICC inconsistency occurs when a composition relationship and an inheritance chain form a cycle that would produce an infinite containment of objects upon instantiation. A first occurrence ICC_1 of this type appears in the inheritance chain `Vehicle` ← `Boat` ← `Amphibious Vehicle`. The second inconsistency ICC_2 occurs in the inheritance chain `Vehicle` ← `Car` ← `Amphibious Vehicle`.

A CI inconsistency arises when an inheritance chain forms a cycle. A first occurrence CI_1 can be observed in the inheritance cycle involving the classes `Vehicle`, `Boat` and `Amphibious Vehicle`. The second occurrence CI_2 occurs in the inheritance cycle involving the classes `Vehicle`, `Car` and `Amphibious Vehicle`.

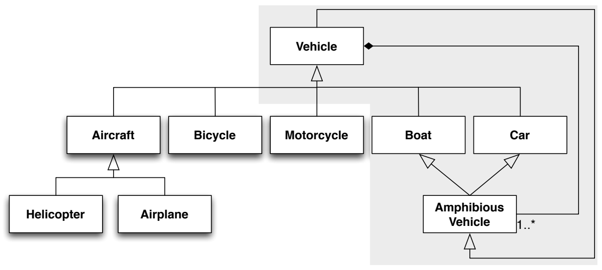

Figure 1: Class diagram with 4 inconsistency occurrences, inspired by [11].

All four inconsistency occurrences share two of the three classes that compose their respective inheritance chains: `Vehicle` and `Amphibious Vehicle`. Due to this overlap, the same resolution action can resolve more than one inconsistency occurrence. For example, removing the composition relationship between `Vehicle` and `Amphibious Vehicle` solves the two inconsistency occurrences ICC_1 and ICC_2. Removing the inheritance relationship between `Boat` and `Amphibious Vehicle` solves the two inconsistency occurrences ICC_1 and CI_1. This clearly illustrates that, in order to resolve model inconsistencies in an optimal way, it is important to consider all inconsistencies simultaneously.

Using the example of Figure 1, we will illustrate how to create a sequence of inconsistency resolution actions with two automated planning approaches: progression planning and regression planning. In both cases, we require as input an initial state (the inconsistent model), a set of possible actions (that change the model) and a desired goal (negation of inconsistencies, see further). Planning requires logic conditions as input, so the whole model environment (*e.g.* model, meta-model, detection rules) is translated into a conjunction of logic literals. In this section, we will use Prolog syntax for explaining how automated planning works.

The (simplified) *metamodel* for class diagrams is expressed below as a set of logic literals. Logic variables start with an uppercase letter. Each model element is referred to by a unique `Id`.

```
class(Id, Name).
generalisation(Id, Label, Child_class, Parent_class).
association_end(Id, Class, Role, UpperMult, LowerMult, Composite).
association(Id, Name, Ass_end_1, Ass_end_2).
```

The **initial state** is expressed as a conjunction of literals, and represents the current world. In our case the initial state will be the inconsistent model. The initial state can be represented either by using the *complete model*, or by using a *partial model* that contains only those elements that are involved in one or more inconsistency occurrences. Below is an example of a *partial model* (conforming to the aforementioned metamodel), containing only the elements that are involved in the inconsistency occurrences, shown in the shaded part of Figure 1.

```
class(c1, vehicle).
class(c5, boat).
class(c6, car).
class(c9, amphibious_vehicle).
generalisation(g4, label4, c5, c1).
generalisation(g5, label5, c6, c1).
generalisation(g8, label8, c9, c5).
generalisation(g9, label9, c9, c6).
generalisation(g10, label10, c1, c9).
association_end(ae1, c9, role1, star, one, non).
association_end(ae2, c1, role2, one, one, yes).
association(a1, ass1, ae1, ae2).
```

The **set of actions** corresponds to the elementary operations (create, modify and delete) of the different types of model elements that can be derived from the metamodel. For each action, a *precondition* needs to specify the conditions that must hold before the execution of the *action*. As an example, the logic rule `can` specified below gives the precondition for `modify_Association_Name`.

```
can(modify_Association_Name(Id, Name, NewName),
    [association(Id, Name, Ass_end_1, Ass_end_2)]) :-
    id(Id),
    string(Name),
    string(NewName),
    NewName \== Name.
```

The *effects* of an action are relationships that express which model elements are added to or deleted from the current state. As an example, the logic rules `adds` and `deletes` below show this for the action `modify_Association_Name`.

```
adds(modify_name_Association(Id, _Name, NewName),
     [association(Id, NewName, Ass_end_1, Ass_end_2)]).
deletes(modify_name_Association(Id, Name, _NewName),
        [association(Id, Name, Ass_end_1, Ass_end_2)]).
```

The **desired goal** is a partially specified state, represented as a conjunction of literals using logic quantification. It specifies the objective we want to reach, namely the absence of model inconsistencies. To achieve this we can use two alternatives: (1) the negation of the inconsistency occurrences; or (2) the negation of the inconsistency detection rules. An inconsistency detection rule is a conjunction of logic literals representing a pattern that, if matched in the model, detects inconsistency occurrences. Below we give an example of the "Inherited Cyclic Composition" detection rule. It only specifies an inheritance chain involving three classes because the planner syntax does not allow to express transitive closure to make the rule more generic.

```
[generalisation(G1, Label1, C, A),
 generalisation(G2, Label2, B, C),
 association(A1, Name, AE1, AE2),
 association_end(AE1, B, Role1, Upper1, one, Composite1),
 association_end(AE2, A, Role2, Upper2, Lower2, yes)]
```

One of the two occurrences in the model that match this detection rule is given below.

```
generalisation(g4, label4, c5, c1).
generalisation(g8, label8, c9, c5).
association(a1, ass1, ae1, ae2).
association_end(ae1, c9, role1, star, one, non).
association_end(ae2, c1, role2, one, one, yes).
```

Using the negation of the inconsistency occurrences in the desired goal will only be able to resolve inconsistency occurrences that have already been identified previously. Using negation of inconsistency detection rules has the advantage that it can be used to detect and resolve inconsistency occurrences at the same time, but suffers from scalability problems (see further). In both alternatives logic negation is used to express the absence of inconsistencies in the resulting model. This implies that we need a planning language that allows the use of disjunction and negative literals in the goal.

A **plan** is a sequence of actions that transforms the initial model into a model that satisfies the desired goal (i.e., a consistent model). A plan is generated automatically by the planning algorithm, without relying on any domain-specific information. Moreover, the generated resolution plan does not lead to ill-formed models (that do not conform to their metamodel) as long as all metamodel constraints are given as part of the problem specification. Two complete resolution plans, containing only two actions, that solve the four inconsistency occurrences of the motivating example are given below:

```
Resolution plan 1 :

1. delete_Generalisation(g5, label5, c6, c1)
2. delete_Generalisation(g4, label4, c5, c1)

Resolution plan 2 :

1. delete_Generalisation(g10, label10, c1, c9)
2. modify_lower_Association_End(ae1, 1, 0)
```

5. Comparative study

5.1. Choice of planners

Since the goal of our article is to compare the performance of progression planning and regression planning for the purpose of automated model inconsistency resolution, we need to select a progression planner and regression planner that are most appropriate for our needs.

To choose a *progression planner*, we surveyed the state-of-the-art on existing planner tools. In 1971, Fikes *et al* [12] developed a formal planning representation language called *STRIPS*. In 1989, Pednault [13] developed a more advanced and expressive language called *ADL*. It allows to use negative literals and disjunction and applies the open world principle. This principle states that unspecified literals are considered as unknown instead of being assumed false.

PDDL [14] is a generic language (*Planning Domain Definition Language*) allowing to represent the syntax of STRIPS, ADL and other languages. Even if PDDL covers all the functionalities of these languages, the majority of planners only implement the STRIPS subset [10]. An important constraint for us is that the planning language needs to support disjunction and negative literals, because the desired goal is expressed as a negation of inconsistency occurrences or rules (see section 4). *FF* (for "Fast-Forward Planning System" [15]) is a heuristic state-space progression planner, and the only one we found to be able to properly deal with negation. To be precise, FF supports the PDDL language with full ADL subset support, including positive and negative literals, conjunction and disjunction, negation, typing, and logic quantification in the desired goal. Therefore, FF is the tool that we have selected for our experiments.

To choose a *regression planner*, we again surveyed the state-of-the-art in planner tools, but did not find a readily available *regression planner* that fit our needs. Therefore, based on the algorithms explained in [16] we started implementing our own regression planner in Prolog, because this logic programming language provides more expressiveness than FF. We baptised the regression planner we implemented *Badger*.

5.2. Experimental setup

Our experimental comparison aims to assess which type of planning algorithms relying on state space search is most appropriate (w.r.t. expressiveness, scalability and other important factors) to be used for resolving inconsistencies in software models. We have carried out a number of experiments with both types of planners presented in subsection 5.1.

All experiments have been performed using a 64-bit Apple MacBook with 2.4 GHz Intel Core 2 Duo processor and 4GB RAM, 2.9GB of which were available for the experiments. In order to remove noise, each experiment was executed 10 times and the average time and standard deviation was computed.

For both considered planners, the generated resolution plans were always complete (i.e., they removed all occurrences of all inconsistencies that were taken into consideration). Typically, there are many ways in which inconsistencies can be resolved. Since both considered planners look for the shortest path in the search space, they always provide a resolution strategy that is *minimal* in the number of actions required to resolve all inconsistencies. As we will discuss in future work, other resolution strategies can be envisaged.

Even if the generated plan removes all selected inconsistency occurrences, it can still introduce new and different inconsistencies in the model. In section 7 we will discuss this issue in more detail.

5.3. Feasibility study

Our first experiments consists of assessing whether the use of both considered planners is at all feasible. We explored the impact of different ways to provide input to the planner, as explained in section 4. The initial state can be specified by giving a *complete model* or a *partial model* containing only those elements that are involved in the inconsistency occurrences (shaded part of Figure 1). The desired goal can either contain a negation of the inconsistency detection rules or a negation of the inconsistency occurrences.

Table 1: Comparison of timing results on the case study of section 4 using both planners. Time is expressed in seconds and the standard deviation is mentioned after the ± sign.

Experiment number	Initial state: Model	Desired goal: Negation of inconsistency	Average time & for FF (in seconds)	Average time for Badger (in seconds)
1	*complete*	*rules*	out of memory	N/A
2	*partial*	*rules*	out of memory	N/A
3	*complete*	*occurrences*	14.84 ± 0.09	0.181 ± 0.003
4	*partial*	*occurrences*	0.268 ± 0.004	0.051 ± 0.003

Table 1 summarises the timing results for each combination of choices using the two selected planners on the class diagram of Figure 1. Table 1 clearly shows that using the negation of *inconsistency rules* for the desired goal gives rise to an out of memory using FF. The negation of *inconsistency rule* cannot be used as desired goal in Badger because this planner requires all the goals to be completely instantiated, while the negation of *inconsistency rule* is based on variables in the goal. Therefore, the remaining experiments only use the negation of *inconsistency occurrences* as desired goal.

5.4. Comparison

To verify how both considered planners perform on larger models, we conducted a series of experiments in which we artificially increased the size of the example class diagram of section 4 in order to assess how this affects the time needed to generate of a minimal resolution plan.

Adding isolated classes to the model.. To illustrate the advantage of using partial models as opposed to complete models as initial state, we reran experiment 3 of Table 1, while artificially augmenting the size of the model by gradually adding a number of isolated classes. The progression planner took more than 5 hours for 20 added isolated classes. When repeating the same experiment with our regression planner, we observe that it outperforms the progression planner with several orders of magnitude (Figure 2). For 20 added isolated classes, the regression planner takes 1.21 seconds.

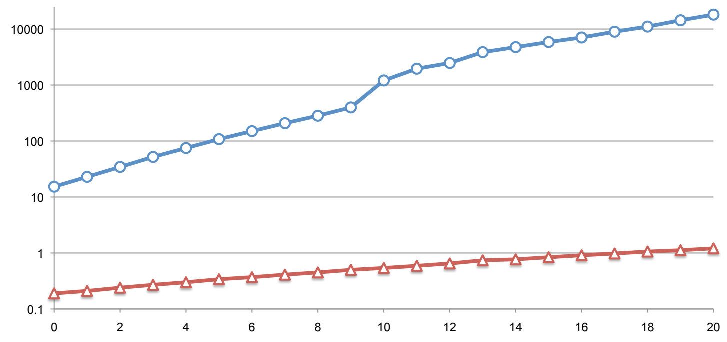

Figure 2: Timing results for progression planning (blue circles) and regression planning (red triangles) using a complete model (experiment 3 of Table 1. The y-axis represents the time in seconds on a logarithmic scale. The x-axis represents the number of isolated classes added to the initial model.

We performed a regression analysis to compare the growth with 5 different models: a linear model, a quadratic model, and exponential model, a logarithmic model and a power curve. Based on the results of this regression analysis, shown in Table 2, we find that the results of FF grow more rapidly (the quadratic and exponential model offer the best fit) than Badger: still the quadratic and exponential model are the best fit but the parameter a is very small, implying that the growth remains very slow, and close to linear, as can be confirmed by a good R^2 value for the linear model.

Table 2: Comparison of regression models on timing results of Figure 2.

Regression model	R^2 value for FF	R^2 value for Badger
linear: $ax + b$	0.769 ($a = 790, 4$)	**0.975** ($a = 0.0493$)
exponential: $b * e^{ax}$	**0.979** ($a = 0.3698$)	**0.994** ($a = 0.0939$)
power: $b * x^a$	0.887 ($a = 2.563$)	0.908 ($a = 0.6531$)
quadratic polynomial: $ax^2 + bx + c$	**0.981** ($a = 80.84$)	**0.999** ($a = 0.0015$)
logarithmic: $a * ln(x) - b$	0.469 ($a = 4494$)	0.757 ($a = 0.3163$)
Conclusion	Exponential or quadratic growth	Very slow growth, very close to linear

Increasing the inheritance chain to the partial model.. We also reran experiment 4 of Table 1 for models of increasing size. As the introduction of isolated classes does not affect the *partial model* used as initial state, the timing results remain *constant*, irrespective of how many isolated classes are added. To assess the effect of an increase of the size of the *partial model* on the time needed to compute a resolution plan, we artificially augmented the size of the model by gradually increasing the length of the inheritance chains involved in the inconsistency occurrences of Figure 1. Figure 3 shows the timing results obtained with the regression planner and the progression planner, after adding between 1 and 8 intermediate superclasses.

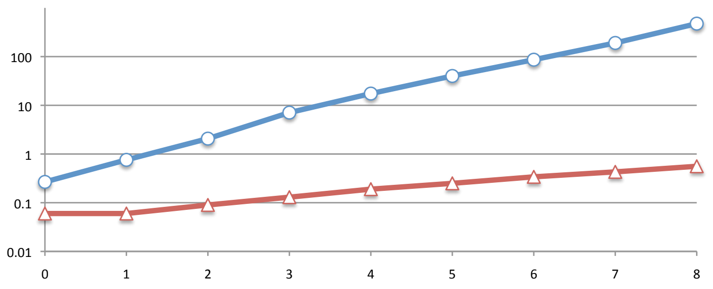

Figure 3: Timing results for adding intermediate superclasses to the partial model using progression planning (blue circles) and regression planning (red triangles). The y-axis represents the time in seconds on a logarithmic scale.

For this experiment, regression analysis reveals that the exponential model is the best one for FF, whereas for Badger the quadratic model is better (though the exponential is also still very good). The detailed results of R^2 values of all analysed regression models are given in Table 3.

Table 3: Comparison of regression models on timing results of Figure 3.

Regression model	R^2 value for FF	R^2 value for Badger
linear: $ax + b$	0.684 ($a = 22.443$)	**0.956** ($a = 0.0701$)
exponential: $b * e^{ax}$	**0.994** ($a = 0.1239$)	**0.991** ($a = 0.3168$)
power: $b * x^a$	**0.947** ($a = 0.1233$)	**0.959** ($a = 1.0849$)
quadratic polynomial: $ax^2 + bx + c$	**0.950** ($a = 6.995$)	**0.999** ($a = 0.0074$)
logarithmic: $a * ln(x) - b$	0.452 ($a = 63.538$)	0.780 ($a = 0.2206$)
Conclusion	Exponential growth	Quadratic (or exponential) growth

Size of the goal.. Finally, we verified whether the number of inconsistency occurrences to be resolved affected the timing results. To achieve this, we restricted the desired goal to generate resolution plans that resolve only 2 or 3 inconsistency occurrences, without affecting the partial model of Figure 1.[1] As we can

[1]We did not do this for 1 inconsistency occurrence only, as it would reduce the size of the partial model, making the results uncomparable with what we found for 2 or 3 inconsistency

see in Figure 4, a reduction of the goal that does not affect the size of the partial model does not have a significant impact on the performance. The growth rate and timing results are still similar to what we found in Figure 3.

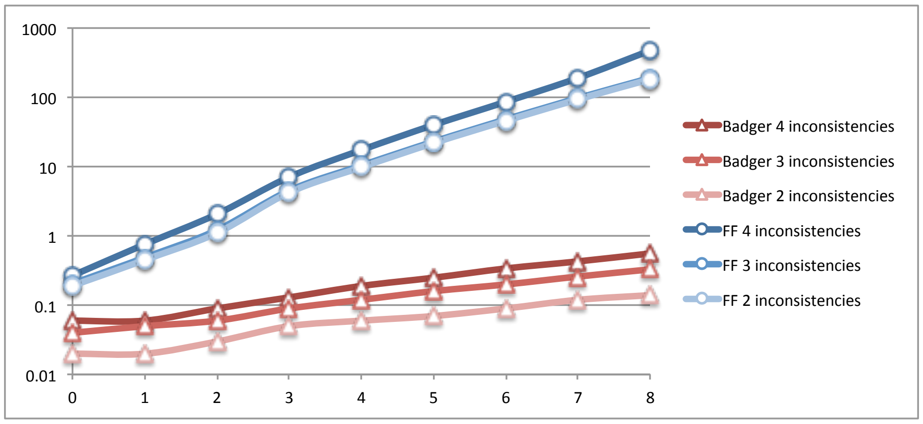

Figure 4: Timing results for progression planning (blue circles) and regression planning (red triangles) for a different number of inconsistency occurrences to be resolved on a partial model. The y-axis represents the time in seconds on a logarithmic scale. The x-axis represents the number of intermediate superclasses added to the initial model.

6. Discussion of Results

From all these experiments, we can observe that the type of planner used significantly affects the results. We compared the progression planner FF with our own regression planner. While FF does the job, it suffers from scalability and has very poor timing results.

We assume that the underlying reason is that FF appears to be optimized for a range of problems in which the search tree is typically narrow and deep, and in which negation is seldomly needed. For the purpose of model inconsistencies, we require negation in the desired goal, and deal with search trees that are wide (there are many actions to consider at each step) and shallow (the total number of actions in the resolution plan is roughly proportional to the number of inconsistency occurrences that need to be resolved).

Regression planners appear to perform significantly better for this type of problem because they restrict the search space by excluding many irrelevant actions. Moreover, since we implemented a regression planner ourselves in Pro-

occurrences.

log, we can still optimise it further to take into account specificities about the problem domain.

Let's discuss a number of points that require more thorough investigation. Using inconsistency rules (as opposed to inconsistency occurrences) in the desired goal leads to out of memory problems, as the search space becomes too big. Nevertheless, a distinct advantage of inconsistency rules is that it enables detection and resolution of inconsistencies at the same time. An approach based on inconsistency occurrences requires a preliminary phase in which the inconsistency occurrences have been detected.

Providing the complete model as initial state may not be realistic, as it gives out of memory errors because the search space becomes too big. Using a partial model as input resolves this problem (at least in the experiments we carried out).

Our experiments were based on a single case study (a small class diagram with 4 inconsistency occurrences of two different types). The results of our experiments may therefore be biased. A proper validation would require a wide range of models, of varying sizes and using different metamodels. Note that the automated planning approach does not depend on a particular metamodel, so it is easy to apply it to the resolution of different types of models.

From a usability point of view, it is fairly straightforward to write a convertor that automatically transforms models (and metamodels) into the logic format required as input by the planner, and to output the resolution plan in the form of a model transformation that is able to correct the inconsistent model. This would avoid users of the approach to learn a new language.

Most automated planners suffer from lack of expressiveness. For example, FF was unable to express transitive closure, primitive types and numbers. Our regression planner in Prolog does not have this problem. Of course, one always needs to find a trade-off between more expressiveness and better performance. The more expressive, the wider range of inconsistencies that can be detected and resolved, but the longer it may take to find a resolution plan (if at all).

7. Future Work

The current representation we used for specifying models (see Section 4) was metamodel dependent: for each metamodel element (e.g. `class`), a Prolog predicate was defined, and model elements were represented as facts using this predicate (e.g., `class(c1,vehicle)`). We are changing this internal representation to make it metamodel independent, based on the work of Blanc et al. [17]. They represent models as sequences of elementary model construction operations, parameterised by the type of model element (e.g., `add(class,c1)`, `addProperty(c1,name,vehicle)`). Besides making the approach metamodel-independent, it has several more advantages. First, the set of actions needed for a particular metamodel can be generated automatically, taking into account the metamodel constraints. Secondly, the operation-based representation of models uses basically the same format as the generated resolution plans. As such, applying a resolution plan to a model becomes trivial. Thirdly, it will allows us to use their *Praxis* framework offering an integration with modeling tools. Praxis is an Eclipse plugin that uses EMF models and comes with: (i) a peer-to-peer model editing framework [18]; (ii) an incremental inconsistency detection tool [17]; and (iii) a model generator [19]. The incremental detection tool will allow us to come up with an iterative resolution approach: whenever a resolution plan is proposed that introduces new inconsistencies, the detection tool will come up with new inconsistency occurrences that can be resolved subsequently. The model generator will allow us to assess the scalability of our approach, by enabling the generation of useful models of arbitrary large size to be used for our experiments.

Automated planners compute a resolution plan that is minimal in the number of actions to carry out to resolve all inconsistency occurrences. This does not mean that the resolution plan is also unique. More generally speaking, a minimal plan may not always be the most appropriate solution. In order to investigate what is the most appropriate resolution plan, we should generate all possible resolution plans (up to a certain size), and let the designer choose the most appropriate one (based on some criteria to be defined, like minimality, conceptual distance, monotonicity). The challenge here is to define the right criteria to decide when a plan is more appropriate than another. User studies will be needed to address this aspect.

In the planner approaches as currently proposed, we make use of sets of primitive actions. In practice, and for larger plans, however, it is likely that certain sequences or combinations of actions occur much more frequently than others. As such we need to find a way to deal with composite actions, and take them into account when proposing the most appropriate resolution plan.

As a final avenue of further research, Harman [20] advocates the use of search-based approaches in software engineering. This includes a wide variety of different techniques and approaches such as metaheuristics, local search algorithms, automated learning, genetic algorithms. We believe that these techniques could be applied to the problem of model inconsistency management, as it satisfies at least three important properties that motivate the need for search-based software engineering: the presence of a large search space, the need for algorithms with a low computational complexity, and the absence of known optimal solutions.

8. Conclusion

In this article we addressed an important problem in the field of *automated software evolution*, namely the automation of model inconsistency resolution. For this purpose, we used *automated planning*, a logic-based approach originating from artificial intelligence. We are not aware of any other work having used this technique for this particular purpose. We investigated and compared two variants of automated planning: *progression* planning that performs a forward state space search, and *regression* planning that searches backwards. We selected (resp. implemented) a tool for each approach to compare their efficiency.

Our results show that a regression planner performs significantly better than a regression planner. In the future, we will continue to improve this approach to study and select the most appropriate resolution plan for a wide variety of models.

Acknowledgements

This work has been partially supported by (i) the F.R.S. – FNRS through FRFC project 2.4515.09 “Research Center on Software Adaptability”; (ii) research project AUWB-08/12-UMH “Model-Driven Software Evolution”, an *Action de Recherche Concertée* financed by the *Ministère de la Communauté française - Direction générale de l’Enseignement non obligatoire et de la Recherche scientifique, Belgium*; (iii) Avec le soutien de Wallonie - Bruxelles International et du Fonds de la Recherche Scientifique, du Ministère Français des Affaires étrangères et européennes, du Ministère de l’Enseignement supérieur et de la Recherche dans le cadre des Partenariats Hubert Curien.

Vitae

Tom Mens obtained the degrees of Licentiate in Mathematics in 1992, Advanced Master in Computer Science in 1993 and PhD in Science in 1999 at the Vrije Universiteit Brussel. He was a postdoctoral fellow of the Fund for Scientific Research – Flanders (FWO) for three years. In 2003 he became a lecturer at the Université de Mons, where he founded and directs a research lab on software engineering. Since 2008 he is full professor. His main interest lies in the underlying foundations of, and tool support for, evolving software. He published numerous peer-reviewed articles on this topic in international journals and conferences. He has been co-organiser, program committee member and reviewer of international symposia and workshops on model-driven software engineering and software evolution. He is involved in several interuniversity research projects and networks, and is founder and director of the ERCIM Working Group on Software Evolution. In 2008 he co-edited the Springer book “Software Evolution” with S. Demeyer. In 2011 he was PC chair of CSMR 2011.

Ragnhild Van Der Straeten is a postdoctoral researcher at the Vrije Universiteit Brussel, Belgium. She has been working in the area of MDE for more than seven years. She obtained her PhD at the Vrije Universiteit Brussel (co-directed by Tom Mens) on the topic of inconsistency management in MDSD. After that, she has published several scientific articles and a book chapter on inconsistency management in MDSD, focussing on the use of formal method support. She has been involved in research projects related to model-driven engineering.

Jorge Pinna Puissant is a PhD student of Tom Mens, working in a research project on Model- Driven Software Evolution. He studies the use of artificial intelligence techniques, in particular automated planning techniques, to analyse model inconsistencies.

References

[1] R. Van Der Straeten, T. Mens, S. Van Baelen, Challenges in model-driven software engineering, in: M. Chaudron (Ed.), Models in Software Engineering, Vol. 5421 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 35–47.

[2] G. Spanoudakis, A. Zisman, Inconsistency management in software engineering: Survey and open research issues, in: Handbook of Software Engineering and Knowledge Engineering, World scientific, 2001, pp. 329–380.

[3] T. Mens, R. Van Der Straeten, M. D'Hondt, Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis, in: Proc. Int'l Conf. Model Driven Engineering Languages and Systems, Vol. 4199 of Lecture Notes in Computer Science, Springer, 2006, pp. 200–214.

[4] C. Nentwich, W. Emmerich, A. Finkelstein, Consistency management with repair actions., in: Proc. 25th Int'l Conf. Software Engineering, IEEE Computer Society, 2003, pp. 455–464.

[5] T. Mens, R. Van Der Straeten, Incremental resolution of model inconsistencies, in: Algebraic Description Techniques, Vol. 4409 of Lecture Notes in Computer Science, Springer, 2007, pp. 111–127. doi:10.1007/978-3-540-71998-4_7.

[6] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, H. Mei, Supporting automatic model inconsistency fixing, in: Proc. ESEC/FSE 2009, ACM, 2009, pp. 315–324.

[7] M. A. Almeida da Silva, A. Mougenot, X. Blanc, R. Bendraou, Towards automated inconsistency handling in design models, in: Proc. CAiSE 2010, Lecture Notes in Computer Science, Springer, 2010.

[8] A. Egyed, E. Letier, A. Finkelstein, Generating and evaluating choices for fixing inconsistencies in UML design models, in: Proc. Int'l Conf. Automated Software Engineering, IEEE, 2008, pp. 99–108.

[9] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 2002.

[10] S. Jiménez Celorrio, Planning and learning under uncertainty, Ph.D. thesis, Universidad Carlos III de Madrid (2010).

[11] R. Van Der Straeten, Inconsistency management in model-driven engineering: an approach using description logics, Ph.D. thesis, Vrije Universiteit Brussel (2005).

[12] R. Fikes, N. J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, in: Proc. Int'l Joint Conf. Artificial Intelligence, 1971, pp. 608–620.

[13] E. P. D. Pednault, ADL: Exploring the middle ground between STRIPS and the situation calculus, in: Proc. Int'l Conf. Principles of Knowledge Representation and Reasoning, 1989, pp. 324–332.

[14] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, PDDL — the planning domain definition language., Tech. Rep. DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut (1998).

[15] J. Hoffmann, B. Nebel, The FF Planning System: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.

[16] I. Bratko, Prolog programming for artificial intelligence, Addison-Wesley, 2001.

[17] X. Blanc, A. Mougenot, I. Mounier, T. Mens, Detecting model inconsistency through operation-based model construction, in: Proc. Int'l Conf. Software Engineering (ICSE), Vol. 1, 2008, pp. 511–520.

[18] A. Mougenot, X. Blanc, M.-P. Gervais, D-praxis: A peer-to-peer collaborative model editing framework, in: Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems, DAIS '09, Springer-Verlag, 2009, pp. 16–29.

[19] A. Mougenot, A. Darrasse, X. Blanc, M. Soria, Uniform random generation of huge metamodel instances, in: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA '09, Springer-Verlag, 2009, pp. 130–145.

[20] M. Harman, Search based software engineering, in: Computational Science - ICCS 2006, Vol. 3994 of Lecture Notes in Computer Science, Springer, 2006, pp. 740–747.