Resolving Model Inconsistencies with Automated Planning

Jorge Pinna Puissant, Tom Mens
Université de Mons
20 Place du Parc
7000 Mons, Belgique
{jorge.pinnapuissant,
tom.mens}@umons.ac.be

Ragnhild Van Der Straeten
Vrije Universiteit Brussel
1050 Brussel, Belgium
Université Libre de Bruxelles
1050 Bruxelles, Belgique
rvdstrae@vub.ac.be

ABSTRACT
Various approaches have been explored to detect and resolve software model inconsistencies in a generic and scalable way. In this position paper, we outline our research that aims to use the technique of *automated planning* for the purpose of resolving model inconsistencies. We discuss the scalability results of the approach obtained through several stress-tests and we propose several alternatives to the automated planning approach.

Keywords
inconsistency resolution, UML models, automated planning, scalability

1. INTRODUCTION
In *model-driven software engineering (MDE)* [24, 26], model inconsistencies inevitably arise, because a (software) system description is composed of a wide variety of diverse models, some of which are developed and maintained in parallel, and most of which are subject to continuous evolution. Our research focuses on the *resolution* of inconsistencies. The inconsistency resolution activity is divided into the following steps: (1) *Select* the inconsistencies that need to be resolved; (2) Identify possible *resolution plans* to resolve the selected inconsistencies; (3) Perform a *cost-benefit analysis* of the implementation of each of these resolution plans; (4) Select and apply resolution actions, based on the previous choices [25]. We focus on how to automate step (2) of the inconsistency resolution activity: identification of possible resolution plans. To do this, we propose to use the *Automated Planning* technique from the Artificial Intelligence domain.

In this article we give an overview of different automated planning techniques (Section 3). Based on a simple case study (Section 4.1) we present an approach using a forward-chaining heuristic planner to resolve inconsistencies (Section 4.2). One of our requirements is that the time required for resolving inconsistencies has to be sufficiently small so as not to disturb the designer in his/her work. Therefore, we investigate the scalability of the approach to larger software models (Section 4.3). Based on these results we discuss ways to improve the scalability of the proposed technique (Section 4.4). We also discuss alternatives to automated planning that may be more appropriate (Section 5).

2. PROBLEM STATEMENT
State-of-the-art approaches on inconsistency resolution exhibit several problems. In [18], resolution rules are specified manually, which is an error-prone process. Automatic generation of inconsistency resolution actions would resolve this problem. This is what is done by Nentwich et al. [19], 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 may cause problems when inconsistencies and their resolution are interdependent [17]. An additional problem is the interaction of the resolutions with the syntactical constraints imposed by the modelling language. Xiong et al. [28] define a language in which it is possible to specify the inconsistency rule and the possibilities to resolve the inconsistencies. This requires inconsistency rules to be annotated with resolution information. Almeida da Silva et al. [1] propose an approach to generate resolution plans for inconsistent models. The approach is based on the extension of inconsistency detection rules with information about the causes of the inconsistency, and on the use of generator functions, which are manually written and are used to generate resolution actions. 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. [6] 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 are dependent on the modelling language, i.e., they take into account the syntactical constraints of the modelling language and they only consider the impact of one consistency rule at a time. Furthermore these choice generation functions need to be implemented manually.

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. In this article we investigate the use of *Automated Planning* for this purpose.

3. PLANNING TECHNIQUES

Automated planning is a technique coming from artificial intelligence research. It aims to create *plans*, which are 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.

Classical planning is an automated planning subset that aim to find a sequence of actions that reaches a desired state in a finite, static, deterministic and fully observable world. 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.

Fikes *et al* [7] developed, in 1971, a formal planning representation language called *STRIPS* (*STanford Research Institute Problem Solver*). In 1989, Pednault [21] developed a more advanced and expressive language called *ADL* (*Action Description Language*, not to be confused with Architecture Description Language). ADL has an improved expressiveness compared to STRIPS. In particular, ADL applies the open world principle: unspecified literals are considered as unknown instead of being assumed false. ADL also allows to use negative literals and disjunction, whereas STRIPS only allows positive literals and conjunctions. In recent years a standard PDDL (*Planning Domain Definition Language*) [10] has been developed for the *International Planning Competition* (IPC) of the *International Conference on Artificial Intelligence Planning and Scheduling* (ICAPS). PDDL is a generic 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 [14]. The most recent version of PDDL is version 3.0 [9]. This language is used in the competition to compare the benchmarks of different planning approaches [23].

Two main approaches exist to solve classical planning problems [14]: (1) translating the planning problem into a problem that can be solved by a different approach (e.g. a boolean satisfiability problem, a constraint satisfaction problem, or a model checking problem); (2) generating a search space (which can be either a state space, a plan space, or a planning graph) and looking for a solution plan in this space. We will focus on this second approach only. Depending on the direction in which the state space is traversed to look for a solution, we can distinguish between:

Progression planning is a *forward search* that starts in the initial state and tries to find a sequence of actions that reaches a goal state. The problem of this algorithm is that it does not exclude irrelevant actions. An action is considered relevant if it can achieve the goal or one of the conjuncts of the goal.

Regression planning is a *backward state-space search* that starts in the goal state and searches a sequence of actions that reach the initial state. This algorithm avoids the problems of the previous one by working only with relevant actions. The problem of this algorithm is that it is not always obvious to find a possible predecessor of an action.

Another distinction can be made between *total-order* and *partial-order* planning. With the former approach, the set of actions that composes the strategy found by the algorithm is strictly linear and ordered from the initial state to the goal. This category of algorithms cannot execute different actions simultaneously and cannot take advantage of the subdivision of a goal. Instead, *partial-order* planning (POP) explores the plan-space without committing to a totally ordered sequence of actions. POP works back from the goal to the initial state and it can place two actions into a strategy without specifying which comes first. As a result, these actions can be executed in parallel and their order is unimportant because they achieve different sub-divisions of the goal [22, 23]. Neither total-order nor partial-order is efficient without a good heuristic function that estimates the distance from a state to the goal.

Many planners exist that implement some variant of a planner algorithm. In this article we use the heuristic state-space progression planner called *FF* (for "Fast-Forward Planning System" [12, 13]). It is considered by [23] as the "The most successful state-space searcher", and was awarded for *Outstanding Performance* at the AIPS 2000 planning competition and *Top Performer* at the AIPS 2002 planning competition. FF has been chosen not only because of its performance, but also because it uses PDDL language with full ADL subset support, including positive and negative literals, conjunction and disjunction, negation, typing, and logic quantification in the desired goal. This is crucial to our approach, as will be explained in the next section.

4. AUTOMATED PLANNING IN ACTION

4.1 Case Study

Design models can be of different types (*e.g.* UML, Petri nets, feature models, business process models). In this article we restrict ourselves to UML class diagrams [20]. They can suffer from many kinds of inconsistencies, such as structural and behavioural inconsistencies. Figure 1 illustrates a simple class diagram containing two structural inconsistency occurrences of type "inherited cyclic composition" and two occurrences of type "cyclic inheritance" [27].

An inherited cyclic composition inconsistency arises when a composition relationship and an inheritance chain form a cycle that would produce an infinite containment of objects upon instantiation. Both occurrences, `ICC1 and ICC2`, of this inconsistency in Figure 1 arise with the same composition relationship, between `Vehicle` and `Amphibious Vehicle`, but with different inheritance chains. The first occurrence `ICC1` appears in the inheritance chain `Vehicle` ← `Boat` ← `Amphibious Vehicle`. The second inconsistency `ICC2` occurs in the inheritance chain `Vehicle` ← `Car` ← `Amphibious Vehicle`. A cyclic inheritance inconsistency arises when an inheritance chain forms a cycle. Figure 1 has two occur-

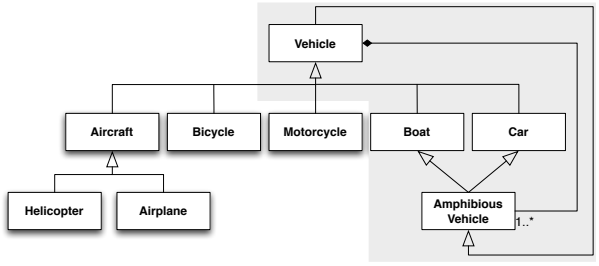


Figure 1: Class diagram with 4 inconsistency occurrences, inspired by [27].

rences `CI1` and `CI2` of this type of inconsistency. The first occurrence `CI1` forms an inheritance cycle that involves the classes `Vehicle`, `Boat` and `Amphibious Vehicle`. The second occurrence `CI2` forms an inheritance cycle that involves the classes `Vehicle`, `Car` and `Amphibious Vehicle`.

All four aforementioned inconsistency occurrences share two of the three classes that compose their respective inheritance chains: `Vehicle` and `Amphibious Vehicle`. Because of this overlap, it is possible to resolve more than one inconsistency occurrence with the same resolution action. For example, removing the composition relationship between `Vehicle` and `Amphibious Vehicle` solves the two inconsistency occurrences `ICC1` and `ICC2`. Removing the inheritance relationship between `Boat` and `Amphibious Vehicle` solves the two inconsistency occurrences `ICC1` and `CI1`. This clearly illustrates that, in order to resolve model inconsistencies in an optimal way, it is important to consider all inconsistencies simultaneously. In [17, 18], the impact of dependencies between model inconsistencies and their resolution actions were studied using the notion of critical pair analysis of graph transformation rules.

4.2 Planning for Inconsistency Resolution

Using the example of Figure 1, we illustrate how to create a sequence of inconsistency resolution actions with automated planning. We require as input an initial state (the inconsistent model), a set of possible actions (that change the model) and a desired goal (the consistent model). 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. The syntax of PDDL is Lisp-like. Each logic literal is a tuple represented between parentheses. The tuple starts with the name of the literal, followed by pairs of variable names and their type (separated by a "–"). There are no primitive types in PDDL. More information about the PDDL syntax can be found in [8].

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. We can choose between three different representations of this initial state: (1) using the complete model; (2) using a partial model that contains only those elements that are involved in one or more inconsistency occurrences; (3) using a partial model that contains only those elements that are involved in a single inconsistency occurrence. We exclude option (3) as it only allows us to solve one inconsistency at a time, and it does not take into account any dependency between inconsistency occurrences or their resolution actions.

The metamodel for our class diagram is given below using PDDL syntax. Each metamodel element is represented by a unique `id` through which it can be referred.

```
(Class ?id - class_id   ?name - String)
(Generalisation ?id - g_id   ?label - String
     ?child_class - class_id  ?parent_class - class_id)
(Association_End ?id - ae_id   ?class - class_id   ?role - String
    ?upper_mult - Cardinal   ?lower_mult - Cardinal
    ?composite - Boolean)
(Association ?id - a_id   ?name - String   ?ass_end_1 - ae_id
?ass_end_2 - ae_id)
```

A partial model conforming to this metamodel is given below. It contains only the elements that are involved in the inconsistency occurrences. This is illustrated by 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** that can be performed to change a model are represented in terms of a *precondition* that must hold before the execution and the *action* to execute. In our approach, inspired by [2], the set of actions corresponds to the elementary operations (basically, create, modify and delete) of the different types of model elements that can be derived from the metamodel. These elementary operations, combined with the logic literals of the metamodel, allow us to compute the list of all possible actions. As an example, the specification of `modify_Association_Name` is given below.

```
(:action modify_Association_Name
 :parameters (?id - id   ?name - String   ?ass_end_1 - ae_id
              ?ass_end_2 - ae_id   ?new_name - String)
 :precondition (Association ?id ?name ?ass_end_1 ?ass_end_2)
 :effect (when (not (= ?name ?new_name))
               (and (not (Association ?id ?name ?ass_end_1 ?ass_end_2))
                    (Association ?id ?new_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 that we want to reach, namely a consistent model. To represent this consistent model we can use two alternatives: (1) the negation of the inconsistency detection rules; (2) or the negation of the inconsistency occurrences. An inconsistency detection rule is a conjunction of literals representing a pattern that, if matched in the model, detects inconsistency occurrences.

The inherited cyclic composition inconsistency detection rule using the PDDL syntax is given bellow. Observe that it only specifies an inheritance chain involving three classes. PDDL syntax does not allow to express transitive closure to make the rule more generic.

```
(exists (?a - class_id   ?b - class_id   ?c - class_id)
 (and
   (exists (?g - g_id   ?Label - g_label)
               (Generalisation ?g ?Label ?c ?a))
   (exists (?g - g_id   ?Label - g_label)
               (Generalisation ?g ?Label ?b ?c))
```

```
   (exists (?ae - ae_id   ?role - ae_role
               ?upper - upper_cardinal ?lower - lower_cardinal)
        (Association_End ?ae ?a ?role ?upper ?lower yes))
   (exists (?ae - ae_id   ?role - ae_role
       ?upper - upper_cardinal ?composite - boolean)
        (Association_End ?ae ?b ?role ?upper one_1 ?composite))
))
```

The advantage of using alternative (1) above is that it can be used to detect and resolve inconsistency occurrences at the same time. Alternative (2) will only be able to resolve inconsistency occurrences that have already been identified previously. On the other hand, as we will see later, alternative (1) suffers from severe scalability problems. In both alternatives we use logic negation to express the fact that we do not want inconsistencies in the model. Because negation of the conjunction of literals is used we need a planning approach that allows the use of disjunction and negative literals in the goal. This is one of the main reasons why we have selected FF as a planning tool for our experiments.

The **plan** is a sequence of actions that reaches the desired goal. It is generated automatically by the domain independent planning algorithm. A complete resolution plan that solves the four inconsistency occurrences of the motivating example is shown in Figure 2. Remark that our approach prohibits the generation of a resolution plan that leads to ill-formed models (i.e., models that do not conform to their metamodel).

```
delete_Generalisation :
        (Generalisation g10 label10 c1 c9)
modify_Association_End_Lower_Multiplicity :
        from:   (Association_End ae1 c9 role1 star one non)
        to:     (Association_End ae1 c9 role1 star zero non)
```

Figure 2: Complete resolution plan that resolves all four inconsistency occurrences.

4.3 Scalability Study

There are different ways in which to specify the input for the automated planning algorithm. To specify the initial state, we can either use the complete model or we can restrict the search space by using a partial model that contains only those elements that are involved in the inconsistency occurrences. To specify the desired goal, we can choose between using the negation of the inconsistency detection rules or using the negation of the inconsistency occurrences themselves.

In order to assess which of the above four choices produces the best results, we compared the timing results of each considered possibility. In order to remove noise, each experiment was executed 10 times and the average time and standard deviation was computed. All experiments were carried out on 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 experiment.

The experiments using the complete model as initial state and the negation of the inconsistency detection rules as desired goal and using the partial model as initial state and the negation of the inconsistency detection rules as desired goal, ran out of memory. Using the complete model as initial state and the negation of the inconsistency occurrences as desired goal, the resolution plan of Figure 2 was generated in 14.84 seconds on average with a very low standard deviation of 0.09 seconds. Using a partial model as initial state, and the negation of the inconsistency occurrences as desired goal, the resolution plan of Figure 2 was generated in 0.268 seconds on average, with a standard deviation of 0.004 seconds.

To verify whether the proposed approach scales up to larger models, we have stress-tested both of the successful experiments (using the negation of the inconsistency occurrences as desired goal). Again each experiment was executed 10 times and the average time and standard deviation was computed.

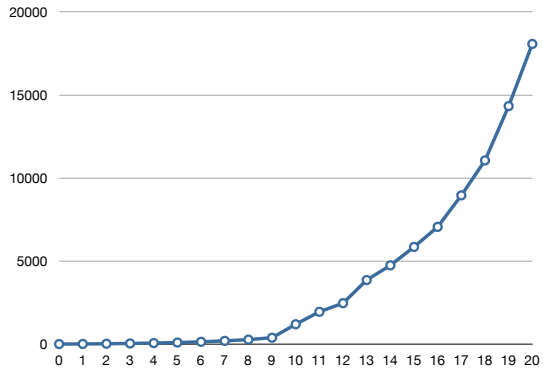
First, we artificially augmented the size of the motivating example of Figure 1 by adding an increasing number of isolated classes to the model (from 1 class to 20 classes). Since these classes are unrelated to the inconsistency occurrences that the algorithm needs to resolve, the algorithm is still able to find the same resolution plan and the partial model is left untouched. However, the time it takes to generate a plan increases as the model size increases.

Figure 3a illustrates the timing results if we use the *complete model* as initial state. It takes only 15 seconds for our initial example, but it takes more than 5 hours for the model with 20 more added classes. A regression analysis reveals an *exponential* relation with coefficient of determination $R^2 = 0.982$, indicating a very good fit of the regression model. Two other candidate regression models we verified had a lower goodness of fit: 0.977 for a quadratic polynomial model and 0.884 for a power curve. These results show that using a complete model as initial state does not scale up to larger models.

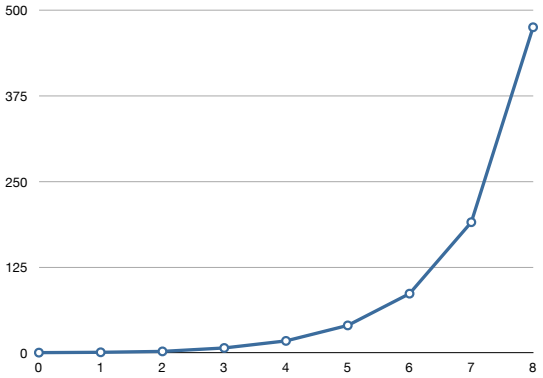
Secondly we studied the timing results when the size of the *partial model* increases. The motivating example of Figure 1 contains an inheritance chain of classes for the two types of considered inconsistencies. We artificially augmented the size of the model by increasing the length of the inheritance chains involved in the inconsistency occurrences. We did this gradually, by adding between one and eight intermediate superclasses, and computing the timing results for each partial model.

Figure 3b shows the timing results of carrying out this experiment. The figure shows a strong increase in time to compute the resolution plan as the size of the partial model increases. The standard deviation was always below 2%, and less than 0.4% on average. A regression analysis reveals an *exponential* growth (with coefficient of determination $R^2 = 0.995$) in the time needed to find a resolution plan. Two other regression models we verified had a lower goodness of fit: 0.927 for a quadratic polynomial model and 0.949 for a power curve.

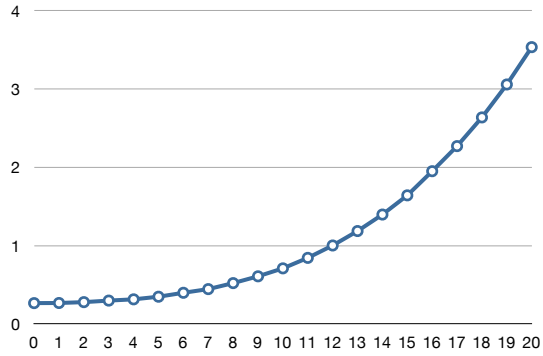
We also verified whether the number of inconsistency occurrences to be resolved affected the timing results. To achieve this, we reduced the desired goal by generating plans that resolve only 2 or 3 inconsistency occurrences, respectively. In all of these cases we found an *exponential* growth in time. We obtained a goodness of fit $R^2 = 0.991$ for resolving 2 inconsistency occurrences, and $R^2 = 0.992$ for resolving 3



(a) Using the complete model as initial state.



(b) Adding intermediate superclasses to the partial model.



(c) Adding class attributes to the partial model.

Figure 3: Scalability timing results (the y-axis represents the time in seconds).

inconsistency occurrences.

Finally, we verified whether the size of the metamodel affects the timing results. To achieve this, we added a new element to the metamodel:

```
(Attribute ?id - attribute_id ?class - class_id
?Name - String ?Type - Type)
```

This requires to add three new actions, to create, modify and delete attributes, respectively. It did not affect the timing results as long as attributes are not used in the initial state and the desired goal.

As a next step, we increased the size of the initial state by adding 1 to 20 attributes to the existing classes of the model. The desired goal was not modified. The standard deviation was 0,7% on average. The results are shown in Figure 3c. For an initial state with 1 attribute added the time was 0.27 seconds. After adding 20 attributes it was 3.5 seconds. A regression analysis revealed a *quadratic polynomial* with a goodness of fit $R^2 = 0.994$. Two other regression models we verified were an exponential model with $R^2 = 0.982$ and a power model with $R^2 = 0.763$.

4.4 Discussion

The exponential timing results obtained through the experiments described in the previous section, indicate that the approach is not usable in practice. Using the approach to resolve inconsistencies one by one would be feasible because the partial model and desired goal will remain relatively small. This is not a good solution, because it does not take full advantage of automated planning. In addition, inconsistency occurrences and their resolution actions are often interdependent. Another important limitation we encountered is the *expressiveness* of the PDDL syntax. It does not offer important features such as transitive closure, primitive types, numbers. In addition, literals cannot be modified (they have to be deleted and added again). A third limitation of our approach is that, currently, we generate only a single resolution plan. The resolution of several inconsistencies can give rise to several different resolution plans, i.e., different sequences of resolution actions leading to possibly different consistent models.

Several improvements to the approach can be envisaged. A first improvement is to adapt the planning algorithm so that it generates several resolution plans among which the model designer could choose. The scalability problem could be adressed by implementing a *domain-specific planner* that can be optimized by making it more specific and more performant for the specific problem we want to tackle. In addition, since we are not constrained by the PDDL syntax, this would solve the problems of expressiveness we encountered. The timing results could be improved by using *regression planning* as opposed to *progression planning* [23], as used by *FF*. Progression planning depends mainly on the size of the initial state and it does not exclude irrelevant actions. Regression planning works only with relevant actions. Because of this, the search space will be significantly smaller. Further experiments are needed to verify whether regression planning will be more appropriate for our needs.

5. BEYOND PLANNING

Since the automated planning approach does not meet our expectations, we would also like to study other techniques coming from the domain of artificial intelligence for the purpose of resolving modeling inconsistencies in an automated way.

Logic-based approaches have been used for different but related purposes in inconsistency resolution. Marcelloni and Akist [15, 16] used *fuzzy logic* to cope with methodological inconsistencies in design models. It remains to be seen whether this approach can be generalised to resolve any kind of model inconsistency. Castro et. al. [5] used *logic abduction* to detect and resolve inconsistencies in source code. Some preliminary results we carried out to apply this approach to resolve inconsistencies in design models appeared promising, but further work is necessary to assess whether the approach scales up and works in practice. Almeida da Silva et al. [1] implemented a Prolog program to generate resolution plans for inconsistent models. The approach is promising but still requires a lot of manual encoded input to specify the generator functions and the causes of the inconsistencies.

Harman [11] advocates the use of search-based approaches in software engineering. This includes a wide variety of different techniques and approaches such as metaheuristics (e.g. variable neighborhood search [3, 4]), local search algorithms, automated learning, genetic algorithms [23]. We believe that these techniques could be applied to the problem of model inconsistency management, because 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.

In order to assess the adequacy of all these different approaches to inconsistency management, there is also an urgent need to define benchmarks allowing to compare them. Such a benchmark should contain at least a set of shared case studies on which to evaluate each approach; as well as a set of clearly identified criteria enabling the comparison of approaches and their quality.

6. CONCLUSION

In this article, we explored the use of *automated planning*, a logic-based approach originating from artificial intelligence, for the purpose of resolving model inconsistencies. We are not aware of any other work having used this technique for this particular purpose. The results of our experiments reveal that the approach is feasible but suffers from various scalability problems. We have discussed ways in which the scalability can be improved. We have also discussed alternative search-based techniques that may deal with inconsistency resolution in a scalable way.

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.

7. REFERENCES

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

[2] X. Blanc, A. Mougenot, I. Mounier, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Proc. Int'l Conf. Software Engineering (ICSE)*, volume 1, pages 511–520, 2008.

[3] J. Brownlee. Variable neighbourhood search. Technical Report CA-TR-20100206-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, February 2010.

[4] G. Caporossi and P. Hansen. Variable Neighborhood Search for Extremal Graphs 1. The AutoGraphiX System. *Discrete Math.*, 212:29 – 44, 2000.

[5] S. Castro, J. Brichau, and K. Mens. Diagnosis and semi-automatic correction of detected design inconsistencies in source code. In *IWST '09: Proceedings of the International Workshop on Smalltalk Technologies*, pages 8–17, New York, NY, USA, 2009. ACM.

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

[7] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *2nd International Joint Conference on Artificial Intelligence.*, pages 608–620, 1971.

[8] A. Gerevini and D. Long. BNF description of PDDL 3.0. http://zeus.ing.unibs.it/ipc-5/, October 2005.

[9] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3 : The language of the fifth international planning competition. Technical report, Department of Electronics for Automation, University of Brescia, Italy, August 2005.

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

[11] M. Harman. Search based software engineering. In *Computational Science - ICCS 2006*, volume 3994/2006 of *Lecture Notes in Computer Science*, pages 740–747. Springer Berlin / Heidelberg, 2006. Workshop on Computational Science in Software Engineering (CSSE'06).

[12] J. Hoffmann. FF: The Fast-Forward Planning System. *The AI Magazine*, 2001.

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

[14] S. Jiménez Celorrio. *Planning and Learning under Uncertainty*. PhD thesis, Universidad Carlos III de Madrid, 2010.

[15] F. Marcelloni and M. Aksit. Leaving inconsistency using fuzzy logic. *Information and Software Technology*, 43(12):725 – 741, 2001.

[16] F. Marcelloni and M. Aksit. Fuzzy logic-based object-oriented methods to reduce quantization error and contextual bias problems in software development. *Fuzzy Sets and Systems*, 145(1):57 – 80, 2004. Computational Intelligence in Software Engineering.

[17] T. Mens and R. Van Der Straeten. Incremental resolution of model inconsistencies. In J. L. Fiadeiro and P.-Y. Schobbens, editors, *Algebraic Description Techniques*, volume 4409 of *Lecture Notes in Computer Science*, pages 111–127. Springer-Verlag, 2007.

[18] T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, October 2006.

[19] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proc. 25th Int'l Conf. Software Engineering*, pages 455–464. IEEE Computer Society, May 2003.

[20] Object Management Group. Unified modeling language: Super structure version 2.1, january 2006.

[21] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332, 1989.

[22] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In *3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 103–114, 1992.

[23] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

[24] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, pages 25 – 31, February 2006.

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

[26] T. Stahl and M. Völter. *Model Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[27] R. Van Der Straeten. *Inconsistency management in model-driven engineering: an approach using description logics*. PhD thesis, Vrije Universiteit Brussel, 2005.

[28] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In H. van Vliet and V. Issarny, editors, *Proc. ESEC/FSE 2009*, pages 315–324. ACM, 2009.