Finding a Path to Model Consistency

Gregory de Fombelle[1,2], Xavier Blanc[2], Laurent Rioux[1], and Marie-Pierre Gervais[2]

[1] Thales Research and Technology
RD 128 F-91767 Palaiseaux Cedex
{gregory.defombelle, laurent.rioux}@thalesgroup.com
[2] Laboratoire d'informatique de Paris 6, Université Paris 6
8 rue de Capitaine Scott – F75015 Paris
{fombelle, xavier.blanc, marie-pierre.gervais}@lip6.fr

Abstract. A core problem in Model Driven Engineering is model consistency achievement: all models must satisfy relationships constraining them. Active consistency techniques monitor and control models edition for preventing inconsistencies, e.g., using automatic errors correction. The main problem of these approaches is that strict enforcement of consistency narrows the modeler's possibilities for exploring conflicting or tradeoff solutions; this is just what temporaries inconsistencies enable. In this article, we propose a hybrid approach capitalizing on active consistency characteristics while allowing the user to edit inconsistent models in a managed mode: at any moment we are able to propose a sequence of modelling operations that, when executed, make the model consistent. The solution consists in defining a set of automatons capturing a sufficient part of the model state space for managing any inconsistent situation. We illustrate this approach on a consistency relationship implied by the application of a security design pattern impacting both class and sequence diagrams of a UML2 model.

1 Introduction

A core problem in Model Driven Engineering is model consistency: all models must satisfy relationships constraining them [2, 9]. This generic definition emphasizes the fact that consistency is a context specific definition, depending on used models, their relationships and their intended uses. An inconsistency is defined as a situation in which models break a consistency rule [3].

There are many consistency techniques. Those techniques often analyze models and report inconsistencies in a static way letting the user trigger checks and correct errors [18, 16, 17]. In this paper we focus on active consistency techniques, enacting at model edition time and interacting with models edition. These techniques aims to make consistency management more "user friendly", e.g., by automatically correcting some errors, forbidding operations or allowing consistency preserving operations.

In the first section we introduce such consistency techniques and raise issues narrowing their usage. Then we present an overview of our approach and illustrate it on a concrete scenario. After a few remarks we conclude the paper.

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 101 – 112, 2006.

2 Consistency Techniques

Consistency by monitoring outlined in [13] is an approach preventing inconsistencies thanks to a checking algorithm executed each time the user requests a model edition operation. Thus consistency rules encoded in the algorithm are impossible to be violated by the model editor. For example, the Objecteering modelling environment does not allow cross package references: each time the user edits a package referencing association, a check is performed and if this consistency rule is broken a dialog box informs the user that the requested operation is not allowed. It is then impossible to break this rule.

The problem with this approach is the strict enforcement of consistency rules. In some cases it would be necessary to relax it. For instance cross package references are allowed in Java and when the user imports Java source code to a Java model (retro-engineering) the user must inactivate consistency checks. In the opposite case the import will fail. But in such a situation, any kind of inconsistencies may be introduced in the model: there should be a balance between strict consistency enforcement and no consistency at all.

Another consistency technique called consistency by construction is also introduced in [13]. It enables automatic completion of models when specific operations are triggered by the user. For example when the user creates an "active class" the consistency engine automatically creates a default state machine and associates it to the class. These methodological consistency rules are often implemented in the model edition user interface. As a consequence, if the user edits a XMI[10] version of its model, he/she can easily break a consistency rule and reload the model into the environment, but without any inconsistency detected.

Constructive approaches can be more complex. In Fujaba, models can be automatically repaired. This works as follows: a background graph rewriting algorithm searches for negative patterns [15] (forbidden patterns) in the graph of objects representing the model. If there is a match (an inconsistency), then the rewriting engine replaces the negative pattern with another predefined correct pattern. This strategy automatically detects inconsistencies and then automatically corrects them. Once again, consistency is enforced, letting no places for inconsistencies, even temporarily. In this case detecting inconsistency and delaying its correction would enable the user to choose between different correct patterns. Furthermore this choice could be performed at his/her convenience, postponing inconsistency resolution at will.

Engels [6, 7] et al. introduce consistency preserving model evolution. Their solution consists in predefining a set of local model transformations rules that have been mathematically proven to preserve a consistency relationship, i.e., protocol consistency and deadlock freedom in this article. The main idea is to preserve consistency incrementally at model edition time, avoiding checking the whole model at each modification. In this approach, no inconsistencies can be introduced if transformation rules are applied correctly, e.g., by respecting a specific order.

The idea of consistency preserving model evolution is attractive since it claims that it is possible to build consistent models incrementally, without running global checks. Unfortunately, they do not describe any mechanisms for managing application order of local model transformations, depending on user awareness of this order. As a

consequence, consistency is not guaranteed, likely resulting in incrementally introduced inconsistencies and requiring an unwanted global model check.

Furthermore enforcing consistency leads to overconstraining modelling activity, frustrating the modeler while he designs solutions, explores multiple alternatives, or does not model at a good precision degree. In [1] authors claim it is impossible, in general to maintain absolute consistency between all perspectives on the system (models) at all times. This position is adopted and reinforced in [5]. Spanoudakis et al. point out positive aspects of inconsistent models, like identification of parts of the system needing further analysis or assistance in specification of alternatives for the development. Finkelstein resumes this situation: "rather than thinking about removing inconsistencies, we need to think about managing inconsistency" [4].

It is clear that active consistency techniques lack such inconsistency management capabilities either by forbidding inconsistencies or by automatically correcting them. But inconsistency-driven correction and prevention reduces the amount of time the user spends in resolution activities. Furthermore, under certain circumstances the user might not have the skills for repairing complex errors. In such a situation those techniques become critical.

We propose to provide active consistency techniques with inconsistencies management capabilities. As a consequence, model edition monitoring and control should not only enable inconsistency prevention and automated correction but should also allow introducing inconsistencies in a managed mode. This implies that an inconsistency no more needs to be repaired synchronously, i.e., blocking user model edition until it is resolved. Instead when an inconsistency is introduced, it is automatically detected but models can still be edited, delaying the automated resolution at will.

3 Principles of Our Approach: Automata for Managing Inconsistencies

Model evolutions are caused by modelling operations triggered by the user or by automated means (i.e., patterns engines, model transformation engines, wizards etc.). While these modifications are performed, models go through a potentially infinite number of different states that are either consistent or inconsistent with regards to a consistency relationship. Before detailing our approach, we introduce model states' spaces and associated concepts.

3.1 Model State Space

We now introduce the theoretical concept of exhaustive model state space **M**, an infinite state transition systems [14] capturing all possible models and model evolutions. In **M,** a state is a model and a transition is a modelling operation.

The figure 1 illustrates the concept of model state space. As we can see it represents two complete (among multiple partial) potential model evolutions between an initial empty model and a model containing two classes linked by an association. Operation o1 means that the user creates a class named C1, operations o2 means that the user creates a class named C2 and operation o3 means that the user creates a

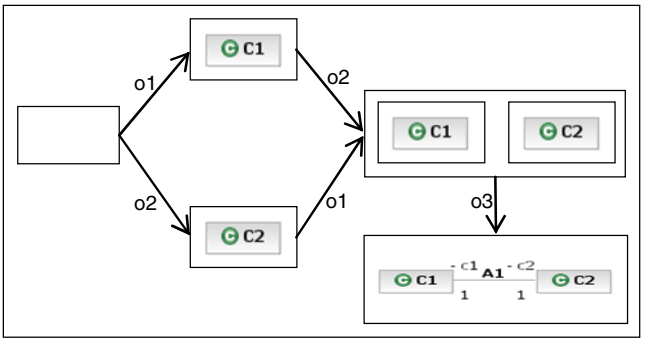

Fig. 1. A simple model state space representation

default association named A1 between C1 and C2. Example modelling operation sequences are <o1>, <o1,o2>, <o2,o1,o3> or <o1,o2,o3>.

The interest of model states lies in their property to validate or not a consistency relationship. The core thesis of our approach is that, given a consistency relationship we are able to specify the sufficient subset of **M** for automatically managing inconsistencies lifecycle. Notice that an inconsistency may exist inside a model (intra-model consistency) or between two or more models, i.e., inter-model consistency [12]. The first step of the solution is on-the-fly detection of transitions switching this model current state:

- from consistent to inconsistent: a new inconsistency is introduced
- from inconsistent to inconsistent: models evolve, but inconsistency remains
- from inconsistent to consistent: an inconsistency is resolved
- from consistent to consistent: models evolve but are consistent

This raises the **M** subset specification issue. Specifying a subpart of **M** implies a clear understanding of M states and transitions. On the first hand, states, i.e., models, are commonly represented as in-memory object data structures, each object being an instance of a meta class defined in the metamodel.

On the other hand transitions are defined as elementary operations on those objects, for instance instantiation of a metaclass, deletion of an object, linking of objects, setting values to object attributes. These operations are provided by the API metamodel repository and are fine grained, in opposition to the o1 operation presented in figure 1. The latter is actually a composite of three "low level" transitions: instanciation of the Class metaclass, setting the default "classname" attribute and linking the newly created object to the Model object.

3.2 Specifying a Subset of M

A direct specification of a subpart M is not conceivable. There are too many states and transitions. Thus it seems not feasible to directly represent a subpart of **M**. We propose three mechanisms for making such a specification feasible.

Transition abstractions save both transitions and states. We have explained that o1 replaces three low level elementary operations; o3 replaces about ten of them. Abstraction of transitions enables to avoid description of intermediate states and transitions.

State abstractions enable description of the sufficient state subpart which might impact the consistency condition. For example we will see in the example introduced in section 4.2 that class attribute descriptions are not necessary since they do not impact the consistency relationship. The language or technique for describing state content is out of the scope of this paper.

State partitioning: we exploit a property of model evolutions that we have observed. From a given state, it is possible to identify and isolate model parts evolving independently of each other. Thus independent model parts may be separated and each subpart evolutions be traced by one automaton. As a result we have a set of automata, each one being responsible for tracing evolution of an independent model part.

In the following abstract example, two automata illustrate such a situation:

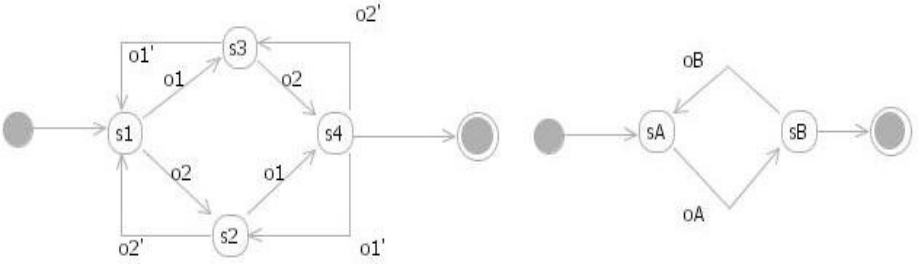

Fig. 2. Two automatons capturing model parts evolutions

Each automaton traces the evolution of a model state space part. For example, if the model is in a composite state given by the two automata states (sA,s1) and the user performs the sequence of model edition operation <o2, o1,o2',o'1>, then only the left automaton will be affected and it will go through states s2, s4, s3 to finally return to s1 state. But at any moment the model can evolve with an oA operation resulting in the appropriate composite model state. We will exemplify it in section 4.3.

3.3 Defining and Managing Inconsistencies with Automata

These automata enable to define consistency or inconsistency in term of relationships over the Cartesian products of automata states. Thus we can mathematically define consistency as a subset C of the Cartesian product of the sets of automata states {A1,…,AK} where Ai is the set of states of Automaton number i. We will illustrate a concrete consistency relation in the section 4.3.

Once consistency defined, the second core idea of our approach is that these automata can be exploited by basic graph algorithms for managing consistent and inconsistent model editions:

- dynamically report to the user that he/she is entering or leaving consistent or inconsistent states.
- computing model edition operations sequences bringing models from inconsistent states to consistent ones. Once this sequence computed it is possible to automatically execute it or propose it to the user.

From an operational point of view our approach consists firstly in automated and incremental detection of inconsistent model states while the user edits models, i.e., on the fly. Secondly it provides users with automatable means for exiting such inconsistent model states. From a theoretical perspective the approach is based on the concept of consistent or inconsistent model state.

4 Running Example

For illustrating our approach, we consider a consistency relationship between two diagrams of a design model constrained by the application of a security design pattern.

4.1 A Security Design Pattern

This security pattern constrains both behavioral and structural properties of the software system. We choose the secure communication security design pattern published by the open security group in their technical guide [8]. The goal of this pattern is to ensure a security policy when two parties need to communicate over a channel that may be subject to security threats. When this pattern is correctly implemented it secures communications against threats by employing protection traffic mechanisms in the communication channel. The structure of this pattern involves three elements exposed in the following figure.

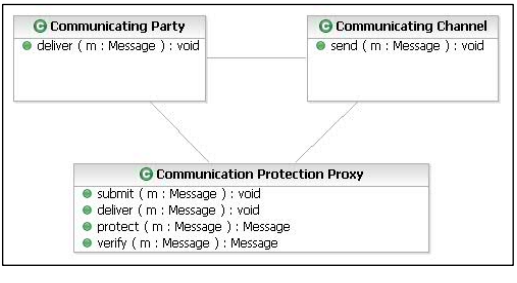

Fig. 3. The secure Communication Design Pattern Structure

The first element is a communication party, e.g., a client or a server. It is the source and/or the destination of messages that are delivered from the communication channel. The second element is the communication channel. It has a send method that, when invoked by parties, transports messages from the sender to the receiver.

Finally the last element is the communication protection proxy. It is responsible for protecting traffic over communication channels. It provides a protection method before sending the message on the channel and a verification method before delivering the message to the target communication party.

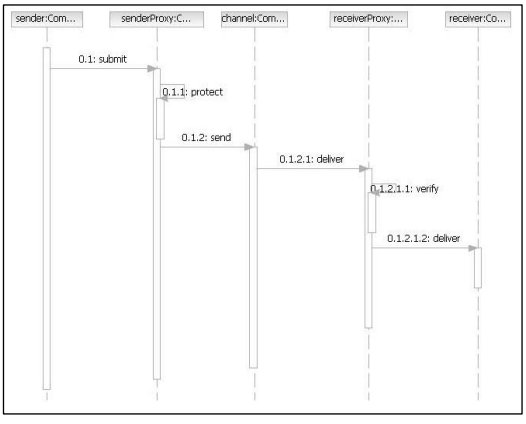

Fig. 4. The Secure Communication Design Pattern Collaboration version 1

This collaboration pattern describes a secured message exchange between a sender communication party and a receiver communication party. First the sender submits a message to its own proxy which protects the data by calling the `protect()` method. Then the proxy sends the message on the channel that delivers it to the receiver proxy. It checks the message by calling the `verify()` method and finally it delivers the message to the final recipient, i.e., the receiver.

4.2 Consistency Scenario

We define consistency of the two diagrams as a configuration in which security aspects of the system are both described in structural and behavioral diagrams. A contradiction may occur if one diagram represents security properties and not the other one. When no one of them represents security properties, then they are consistent because they do not contradict each other [1].

Now suppose we wish to manually secure the simple client server design model below.

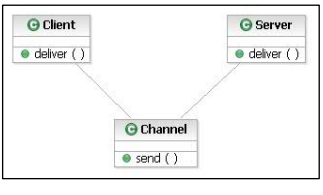

Fig. 5. Unsecured Client Server Structure Diagram

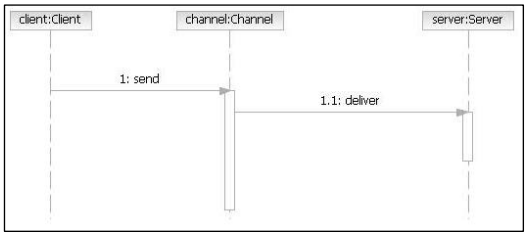

Fig. 6. Unsecured Client Server Sequence Diagram

We define two consistent model states. The first one is when the channel class is stereotyped with a "SecureComm" stereotype and the sequence diagram method calls are intercepted and secured by the proxies. The other one is when the channel class is not stereotyped and the communications are not secured, like in figures 5 and 6.

The user could start introducing a "SecureComm" stereotype on the Channel class. As explained above, both diagrams are consistent if they do not contradict each other. Thus at the moment the user stereotyped the channel class the model is inconsistent and it will remain in this state until the sequence diagram is secured in accordance with the behavioral pattern. We have illustrated a possible model evolution of the sequence diagram from its initial state to its final fully secured state.

Fig. 7. Initial state

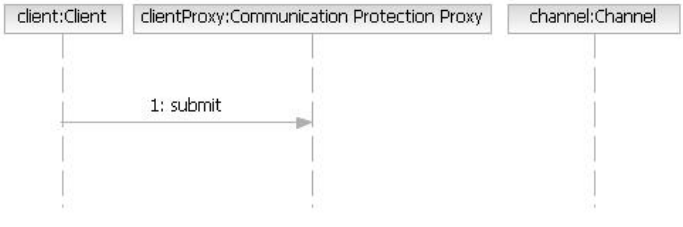

Fig. 8. State 1 of the automaton (cf. fig12.)

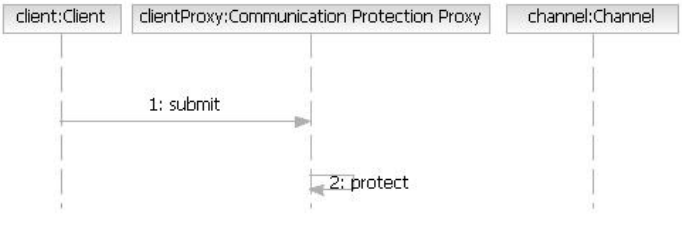

Fig. 9. State 4 of the automaton (cf. fig12.)

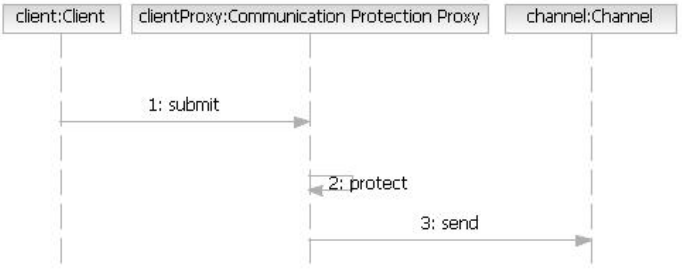

Fig. 10. Secured State of the automaton (cf. fig12.)

From the initial state to the one of figure 8 the user has added the submit method call between the client and the proxy. Then from state of figure 8 to the one of figure 9 the user has added the protect method call. In the last step the user has added the send method call between the proxy and the communication channel.

4.3 Automata Supporting Scenario

In this section we define the automata supporting the scenario, and then we define the consistency relationship.

There are two automata, one responsible for monitoring the evolution of the class diagram and the other monitoring the evolution of the sequence diagram. The first automaton is given in the following figure.

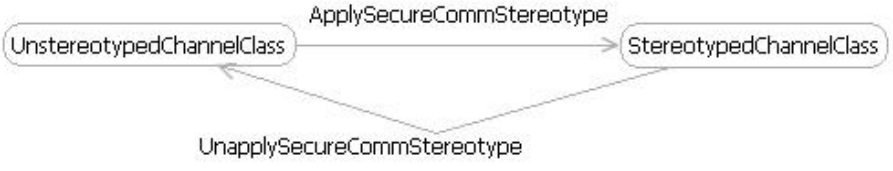

Fig. 11. Automaton 1 for class diagram evolutions

Following the abstraction and partitioning mechanisms this automaton defines two abstract, partial model states respectively representing a “SecureComm” stereotyped channel class and a channel class without this stereotype. The transitions between the two states represent application or deletion of the “SecureComm” stereotype.

The following figure defines the automaton for tracing potential sequence diagram evolutions. Because the full automaton is huge, we have only represented the possible evolutions from the figure 7 to the figure 10. Its structure underlines multiple scenarios for securing these model parts. Each method call addition between two life lines (or one in the case of the protect method call) can be applied independently.

Now we define the consistency relationship. As previously explained the model has two consistent states: one before the application of the pattern and the other when the pattern has been completely applied in the two diagrams. It is possible to formulate this situation in terms of automata states. Indeed the model will be consistent if and only if both automata 1 and 2 are simultaneously in specific states.

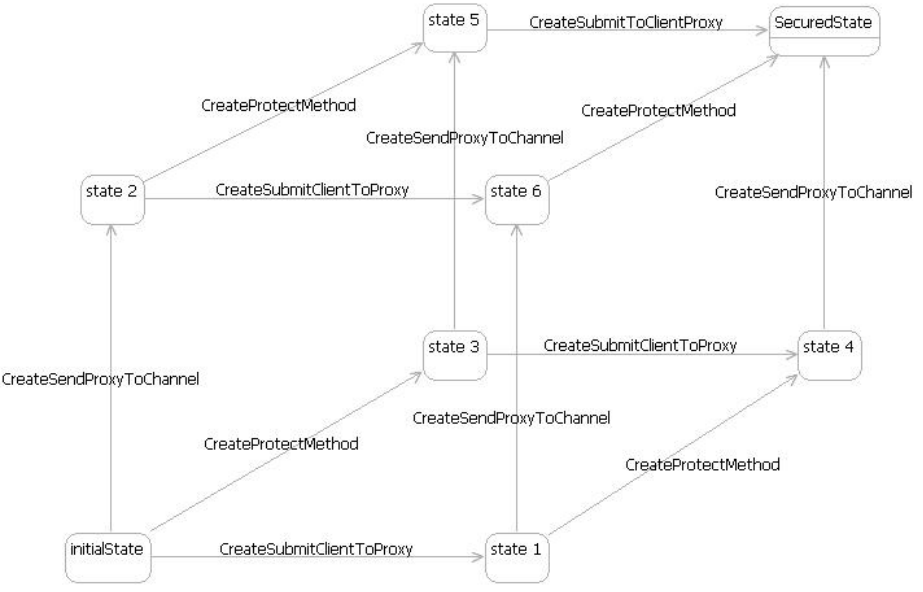

Fig. 12. Automaton 2 for sequence diagram evolutions

Thus we can mathematically define consistency as a subset C of the Cartesian product of automata A1 and A2 state sets ($C \subseteq A1 \times A2$).

A1={*UnstereotypedChannelClass,StereotypedChannelClass*}

A2={*initialState,state1,state2,…,state6,SecuredState*}

C={(*UnstereotypedChannelClass,initialState*),

(*StereotypedChannelClass,SecuredState*)}

In the scenario we have illustrated a possible model evolution of the sequence diagram through different states. This scenario is only one of the possible sequences of modelling operations for building the model. The second automaton illustrates this point clearly: we see that for going from "initialState" to "SecuredState" there are many paths (6 exactly).

Imagine we are in the state where the class is stereotyped and the sequence diagram is in the state illustrated on figure 7. Then, the composite model state captured by our automata is given by the couple (*StereotypedChannelClass, InitialState*). This state is not consistent since it does not belong to the previously defined consistent set C . Thus it is possible to detect an inconsistent state while models evolve. Furthermore it is possible to compute a path in automata for reaching a consistent state; this path is a sequence of modeling operations. In our scenario such a path is for instance <CreateSubmitClientToProxy, CreateProtectMethod, CreateSendProxyToChannel>. If we execute this sequence of operations then it will switch the model to the consistent state (*StereotypedChannelClass, SecuredState*).

4.4 Remarks

A first remark is that there are multiple paths to consistency: it is possible to undo the first modelling operation that leads to an inconsistent state. Here when the model evolves to the state where the channel class is stereotyped, the trivial path to consistency is to undo the stereotyping action.

Some will notice that no inconsistencies will be raised if the user draws a method call between the client and the channel, resulting in an unsecured method call. This is not because our approach cannot deal with this situation. The simple reason is that it is not specified as an inconsistency. But it is possible to specify new automata or to modify existing ones for taking into account this constraint.

These automata have been manually produced following two main principles: abstraction and partitioning. But the complexity of this specification remains high. On this limited but concrete example there are still many states and transitions. We could then head toward automatic production of these automata from high level languages. For instance we believe feasible to generate these automata from simple QVT relation language[11] expressions.

5 Conclusion and Further Works

In this paper we have introduced active consistency techniques, an approach to model consistency enacting at model edition time, automatically correcting errors or preventing modelling operations when they break a consistency condition. Then we explained why their strict consistency enforcement policy should be relaxed. As a result we argued the need for a hybrid approach combining active consistency techniques and "live" inconsistency management capabilities. We detailed some issues of such an approach like how to determine inconsistent model states without running complete model checks.

Main contribution of our approach is to introduce the concept of model state space and to define mechanisms for producing automata tracing all model evolutions which might impact a consistency relationship. These automata enable on-the-fly detection of inconsistencies, specified as a subset of the Cartesian product of their states. Furthermore at each current model state, it is possible to compute paths to consistent states. A path is a sequence of modelling operations and may be automatically executed. But this execution may also be delayed enabling to edit the model while it is inconsistent. To the best of our knowledge this is the first time such an approach is proposed. However in this article this model of model evolution is not formally presented.

Our future works can be divided into theoretical and experimental aspects. At the theoretical level we firstly plan to provide a formalization of this model, a precise definition of modelling operations and model states. Based on these core concepts, we wish to explore the relationships between the automata and the state space of models. At the experimental level we designed an initial architecture and implemented a basic model listener for tracing model evolutions. The latter detects events modifying the model data structure. In the next step we will be able to specify composite modelling operations from atomic ones and detect them while the user edits models.

References

[1] Anthony Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer and Bashar Nuseibeh, *Inconsistency Handling in Multiperspective Specifications* IEEE Transactions on Software Engineering, 20 (1994), pp. 569-577.

[2] Bashar Nuseibeh, Jeff Kramer and Anthony Finkelstein, *A framework for expressing the relationships between multiple views in requirements specification*, IEEE Transactions on Software Engineering, 20 (1994).

[3] Gregor Engels, Jochen M.Küster, *Consistency Management Within Model-Based Object-Oriented Development of Components*, FMCO 2003 proc., LNCS 3188 (2003), pp. 157-176.

[4] Anthony Finkelstein, *A Foolish consistency: Technical challenges in Consistency Management*, *11th International Conference on Database and Expert Systems Applications DEXA 2000*, LNCS, London, UK, 2000, pp. 1-5.

[5] George Spanoudakis, Andre Zisman, *Inconsistency Management in Software Engineering: Survey and Open Research Issues*, in W. S. P. C. Chang S. K., ed., *Handbook of Software Engineering and Knowledge Engineering*, 2001.

[6] Gregor Engels, Jochen Kuster and Reiko Heckel, *Toward Consistency preserving model evolution*, in ACM, ed., *IWPSE* Orlando, 2002.

[7] Gregory Engels, Reiko Heckel, Jochen Kuster and Luuk Groenewegen, *Consistency-Preserving Model Evolution through Transformations*, *UML 2002*, 2002.

[8] Heath, Bob Blakley, *Security Design Patterns*, 2004.

[9] Jean-Louis Sourrouille, Guy Caplat., *Checking UML Model Consistency*, *Workshop on Consistency Problems in UML based software development I*, Dresden, Germany, 2002.

[10] OMG, *MOF 2.0 / XMI Mapping Specification, v2.1*, (2005).

[11] OMG, *MOF 2.0 Query/View/Transformation*, 2005.

[12] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, *Maintaining Consistency between UML models using description logics*, *Workshop on Consistency Problems in UML based software development II*, San Francisco, USA, 2003.

[13] Snoeck M, Michiels C and Dedene G, *Consistency by construction: the case of MERODE*, Workshops ECOMO, OWCMQ, AOIS and aXSD, 2814 (2003), pp. 105-117.

[14] Tel, Gerard, *Introduction to Distributed Algorithms*, Cambridge University Press, 2001.

[15] Wagner, Robert, *A Plug-in for flexible and incremental consistency management*, *Workshop on Consistency Problems in UML based software development II*, San Francisco, USA, 2003.

[16] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Regio, Jean Louis Sourrouille, *Consistency Problems in UML-based Software Development - Workshop proceedings*, *Fifth International Conderence on the Unified Modeling Language and its applications - UML 2002*, 2002.

[17] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Regio, Jean Louis Sourrouille, *Consistency problems in UML-based Software Development II - Workshop proceedings*, *Sixth International Conference on the Unified Modelling Language - the Language and its applications UML 2003*, 2003.

[18] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Regio, Jean Louis Sourrouille, *Consistency Problems in UML-based Software Development III - Understanding and Usage of Dependency Relationships - Workshop proceedings*, *Seventh International Confoerence on UML Modeling Languages and Applications - UML 2004*, Lisbon, Portugal, 2004.