Implementing Consistency Management Techniques for Conceptual Modeling

Raf Haesen and Monique Snoeck
K.U.Leuven
Dept. of Applied Economic Sciences
Naamsestraat 69, B-3000 Leuven, Belgium
{Raf.Haesen,Monique.Snoeck}@econ.kuleuven.ac.be

Abstract

Most software development methodologies justify the use of multiple independent models to represent all aspects at the different stages in the development process. This can make the resulting information system inconsistent. The use of a single model and different views to that model can avoid this problem: all views have to be build according to well-formedness rules for that view and consistency must be checked between the related views. In this way it is possible to obtain a model that reaches a feasible level of validity and completeness. This paper represents different techniques to maintain consistency of one view and the use of the same techniques to enforce and check consistency between the views. We discuss their concrete implementation in a modeling tool, based on the object-oriented domain modeling method MERODE.

1. Introduction

The framework of Lindland, Sindre and Sølvberg for quality-improvement of conceptual models distinguishes itself from previous attempts by not only identifying major quality goals for conceptual models, but also the means for achieving them [15]. As such, the framework contains a core set of quality goals and means, subdivided according to syntactic, semantic and pragmatic quality. With respect to semantic quality, two goals are put forward, i.e. feasible validity and feasible completeness. Validity means that all statements made by the model are correct and relevant to the problem, whereas completeness means that the model contains all the statements about the domain that are correct and relevant. To achieve a feasible level of validity, consistency checking is considered as an important semantic means: it allows verifying the internal correctness of specifications. Inconsistency can be defined as any situation in which a set of descriptions does not obey some relationship that should hold between them [9]. This definition is quite general and encompasses many different types of inconsistencies, for example inconsistencies that arise from the use of synonyms, e.g. 'borrower' and 'user' in the context of a library, inconsistencies between diagrams that present different views of a single model and inconsistencies between documents at different development life cycle stages. In addition to inconsistencies between documents, inconsistencies can also arise in a single document when well-formedness rules for this type of document are violated.

Although the presence of inconsistencies in information system models has not always to be considered as a negative element [18], a quality approach to software development should strive for the maximal avoidance, identification and resolution of inconsistencies, whenever possible. Many software development approaches advocate the use of different modeling techniques for modeling different aspects of the information systems at the different stages of the development process. In this way multiple views on an information system are defined which can be arranged according to two relationships: a horizontal versus a vertical relationship. The horizontal relationship exists between views that belong to the same level of abstraction or development phase, e.g. a class diagram describing the static aspects of a conceptual model and a state chart describing the behavior of one class in this model. The vertical relationship exists between views that model the same aspect, but at different levels of abstraction or different development phases, e.g. a conceptual data model and a logical data model. The type of consistency management between two views, will depend on the way they are related to each other. For horizontally related views, it is mostly assumed that they are built according to the single-model principle [19]. This approach is based on the conception of a single model, for which different views are constructed, and with an automatic or semi-automatic consistency checking among these views. Vertically related views show a construction or refinement dependency: a view on a lower level of abstraction is derived from the corresponding view on the higher level of abstraction. This allows for an automatic construction (or genera-

tion) of (part of) the dependent view. The PIM-PSM transformations of the Model Driven Approach [16] are examples of such construction dependent views: a platform independent model is transformed into a platform specific model according to some transformation rules. Finally one can also identify evolution consistency, which indicates consistency between different versions of a same model [30].

Consistency checking can further be subdivided in syntax checking versus semantic checking and intra-view versus inter-view consistency checking. Many authors make a difference between syntax and semantics although the boundary between the two is sometimes blurred. In [11] D.Harel and B.Rumpe state that as soon as a constraint can be automatically checked, it is a syntactic constraint, while a semantic constraint is formulated in natural language and cannot be checked automatically. This implies that if a modeling language has been well-formalized, more constraints can be automatically checked and are considered to be syntax, whereas for a badly formalized language, almost everything becomes semantic checking. Another view on syntax versus semantics is that syntax refers to the well-formedness of a single diagram or view, whereas semantics refers to the existence of a (mathematical) model defining the "meaning" of a diagram and allowing reasoning and inference on specifications. Intra-view consistency checking refers to checking the constraints that should be satisfied by a single diagram or view. Most of these constraint will be of syntactical nature. But again, there are some rules that can be considered both as a syntactical and as semantic constraint. For example, the formalization of the semantics of inheritance imply that circular inheritance should not be allowed and hence this is transformed to a syntactical constraint that forbids circular inheritance. Inter-view *semantic* consistency checking requires the existence of a common mathematical model for the different views. It is not uncommon to use different mathematical formalisms to formalize different diagramming techniques. However, for the semantical integration of different views, a common model or at least translations between models need to be defined [33].

UML defines a set of nine different views on an information system, but does not explicitly define horizontal or vertical and consistency relationships between those views. Inter-view consistency rules are not explicitly identified, but only implicitly present because meta-classes appear in the meta-model of more than one diagramming technique. To be used effectively, UML must be used in conjunction with a methodology which organizes the views according to some development process and establishes the relationships between the artifacts produced during the development process [12]. In this paper we present a tool for managing specifications according to the methodology MERODE[1].

[1] MERODE stands for Model driven, Existence dependency Relation, Object oriented DEvelopment

MERODE has tackled the problem of consistency by defining a formal syntax and semantics for the different views. In order to keep this manageable, MERODE has drastically reduced the number of views and concepts that can be used. It should therefore be considered as a Domain Specific Language for the conceptual modeling of management information systems. MERODE uses only the class-diagram and the Finite State Machine diagrams of UML, complemented with an proprietary diagram, called an Object-Event Table. As all these diagrams are at the same level of abstraction, we are dealing with horizontal consistency. And as all the constraints have been automated, the consistency checks that are performed would be classified as syntactical checks according to [11]. However, as the semantics of the diagrams have been formalized by means of a common CSP-like process algebra, the consistency checks also ensure the *semantic* consistency of the different views. In addition, the formalization is not achieved by translating the UML-specifications to process algebra in a separate step. Rather, the process algebraic formalism was defined first, and subsequently, concepts from the process algebra have been given a graphical symbol, following the UML conventions. As a result, the approach does not suffer from traceability problems [6]: any reported inconsistency in terms of the underlying process algebra, can be reported in terms of the visual equivalent diagram. The choice for process algebra as semantic formalism implies a behavioral focus on the question of consistency [6].

2. Consistency Management

As explained in [26], three strategies of consistency management can be identified. A first approach is *consistency by analysis*, meaning that an algorithm is used to detect all inconsistencies between a set of deliverables, and a report is generated thereafter for the developers. In this kind of approach the requirements engineer can freely construct the different views. At the end of the specification process or at regular intervals, the algorithm is run against the models to spot inconsistencies and/or incompleteness in the various views. The second approach is *consistency by monitoring*, meaning that a tool has a monitoring facility that checks every new specification against the already existing specifications in the case-tool's repository. Whenever an attempt is made to enter a specification that is inconsistent with some previously entered specification, the new specification is rejected. The advantage of this approach is that the model remains constantly consistent. A third approach is *consistency by construction (or by generation)*, meaning that a tool generates one deliverable from another and guarantees semantic consistency. Whenever specifications are defined in one view, those elements in other views that can automatically be inferred are included in those views. Also

in this approach, the requirements engineer can only define consistent models. This approach is in essence the implementation of a ”cascading create/modify/delete” in analogy with a cascading delete in a DBMS, whereas the monitoring approach is a ”restricted create/modify/delete” in analogy with a restricted delete.

Most existing approaches follow the principle of consistency by analysis. But as discussed in Sect. 5, the inadequacy of UML's formalization makes consistency management for UML very difficult. In this paper we will demonstrate that when a formalized method is available, it is possible to implement all three strategies in a complementary way. We will demonstrate this by means of MERMAID's[2] architecture, a modeling tool based on the object-oriented *domain modeling* method MERODE. Since the aim of the paper is to illustrate the architecture of the tool enabling simultaneously the three consistency management strategies, the methodology is taken “as is”.

3. Consistency Rules

A consistency relationship exists between two views because a third description says it should. Hence, consistency checking requires a clear and formal description of each view and a description of the relationship between these views. For MERODE, the three views have been formalized by means of (process) algebra and a set of consistency rules have been developed to maintain consistency between the three different views [25, 27]. Two of the distinguishing features of MERODE are the particular way of modeling associations between classes and the way of modeling object interaction:

- The class diagram models the static aspects by means of inheritance relationships between classes and 1-to-1 and 1-to-Many associations expressing existence dependency; it is therefore called an existence dependency graph (EDG). An existence dependency relationshipo is equivalent to a UML association where the association end of the master class has cardinality [1..1] and is frozen.

- The object-event table (OET) identifies event types and indicates if and how object types are affected by an event type. If an object type is affected by an event type it will define a method with the event type's name implementing the effect of this event type on objects of this class.

- A set of finite state machines (FSM) per object constitutes the behavioral model.

[2] MERMAID stands for MERode Modeling AID

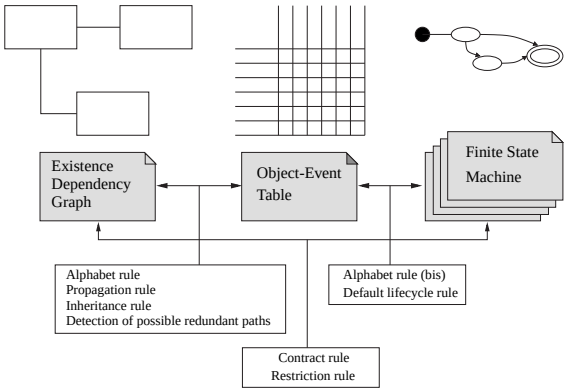

Figure 1. Consistency checking rules

- There are no interaction diagrams or collaboration diagrams as it is assumed that domain objects interact by participating jointly in events. When an event is invoked, some dispatching mechanism (in a layer above the domain layer, not included in the MERODE model) will dispatch the event to all participating domain object types. As a result, *domain* objects will not interact directly with each other by invoking each other's operations.

Figure 1 shows the complete list of *inter*-view consistency checking rules. The application of these rules guarantees that all views are mutually consistent at all time. In addition to these there are numerous rules that guarantee the *intra*-view consistency for each view. These syntax rules are not further discussed. We shortly summarize the implemented *inter*-view consistency rules. The following rules ensure consistency between the existence dependency graph and the object-event table:

1. *Alphabet rule*

   The alphabet rule states that each object should participate in at least two events: one that creates the object and one that destroys the object.

2. *Propagation rule*

   In the class diagram of a MERODE model, all associations should express existence dependency: a class D is existence dependent of a class M if and only if the life of each occurrence of class D is embedded in the life of one single and always the same occurrence of class M [27]. The propagation rule says that the master class M is by default also involved in all event types that the dependent class D is involved in. The methods that the master acquires from its dependents are called acquired methods; methods that are proprietary to a given class are called owned methods.

3. *Inheritance rule*

   Inheritance between a generalization class and a specialization class means that the specialization class "inherits" structure and behavior from the generalization class. In that way the specialization type inherits all the attribute definitions of the generalization type. The inheritance rule states that an object type also inherits all the methods (= the fact of participating in an event type) from its parent object type, either unchanged or by specializing them. Methods that remain unchanged are called inherited methods; if a method is redefined in the specialization class, it is a specialized method. As we need to know for which class an occurrence is created, the creating event type always has to be specialized.

The next rules keep the object-event table and the finite state machines consistent with each other:

1. *Alphabet rule (bis)*

   All finite state machines of an object type must contain all and only the methods in which the object type is involved. This list is obtained by taking all marked event types in the object's column in the OET.

2. *Default lifecycle rule*

   An event can create, modify, or end an object. This is marked in the OET by respectively a 'C', 'M' or 'E' in the corresponding cell. The sequence constraints imposed by the FSM must not violate the default lifecycle of create, modify, end. If we combine these last two rules, a default FSM can be constructed: creating methods bring the object from its initial state to the intermediate state, modifying methods keep the object in the intermediate state, and ending methods put the object in the final state.

The next rules keep the Existence dependency graph and the finite state machines consistent with each other:

1. *Contract rule*

   Whenever two or more that one class participate in the same events, they need to agree on the allowed scenario's (or traces) for these events. The Contract rule induces the creation of a common existence dependent class (= a contract) that model the scenario's both classes will agree on. A more detailed explanation can be found in [25].

2. *Restriction rule* When a class is existence dependent of another class (= the master class), its lifecycle is embedded in the master's life cycle. As all objects run concurrently, the instances of the dependent class will always run concurrently with an instance of the master class. The restriction rule verifies if all scenarios specified by the dependent class are accepted by the lifecycle definition of the master class. This rule partially ensures deadlock-freedom of the specifications and ensures that for each methods there exists at least one scenario where this method can be triggered. This latter condition is called alphabet preservation: it proves that there are no superfluous methods that can be removed from the system without affecting its functionality. In [5] we proved that these binary checks are not sufficient: this rule has to be complemented by an overall deadlock and alphabet preservation checking procedure.

4. Implementation Architecture of MERMAID

4.1. Architecture

The implementation architecture of MERMAID follows the basic principles of MERODE (the proof of the pudding is in the eating...). The methodology's meta-model acts as the domain model and defines the object types that appear in the three types of diagrams and the associations between these object types. The domain objects or meta-model objects do not interact directly with each other, but participate in event types. MERMAID being a requirement engineering tool, the meta-event types are the insertion, update or deletion of a requirement statement such as the definition of a new domain object type, the addition of a business rule, the deletion of an association, etc. Events are triggered by means of the user interface and passed to an event handling layer which dispatches the event to all concerned meta-model objects.

The main event handler is split into three components: the create event handler handles statement insertion, the delete event handler takes care of statement deletion and the update event handler is responsible for all requirements modifications. All public methods in these event handling classes can be called from the user interface in the topmost layer.

All features in the event handlers ultimately lead to a call to a descendant of the class COMMAND, which implements the Command pattern [10]. The choice of this construct can be motivated for different reasons: the addition, deletion and modification of requirement statements are all requests that have a reference to one or more meta-model objects. The deferred top class has features to execute and undo the command, to reverse the effects of the called command. Executed commands are stored in a history list, so unlimited undoing and repeating of commands is supported.

The layered software architecture of MERMAID is represented in Fig. 2. It follows one of the important architectural paradigms of object-oriented programming: each layer re-

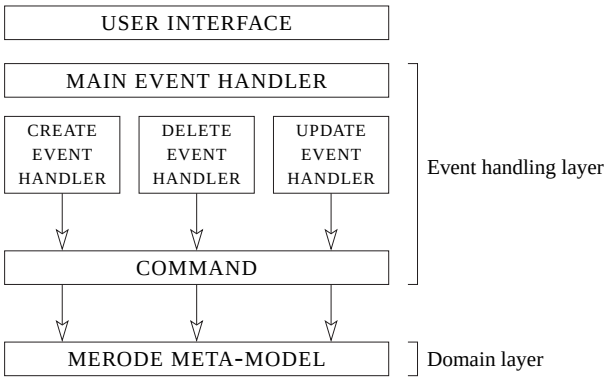

Figure 2. Software architecture of MERMAID

lies on functionalities of its own or on features of the layers below. In this way a high level of abstraction is obtained: the stable lower levels are not influenced by changes in the higher levels [24].

4.2. Consistency Checking

In order to ensure the consistency of specifications we have to make sure that all meta-model objects are created and stored by following the well-formedness rules of each view and the consistency rules that define the inter-view consistency. A few examples of those rules are:

- Domain object types and event types need to have a valid and unique name.
- The creation of a method requires the existence of a (previously created) domain object type and an event type for which the method is implemented.
- An inherited method has a mandatory reference to the parent's method it originates from and to the inheritance relationship that caused the inheritance.
- A specialized method, which is also a special type of method, has a mandatory link to the inherited method which it specializes.

The basic principle applied in MERMAID is consistency by monitoring, but the other two consistency management techniques are also implemented. Therefore MERMAID uses four implementation techniques: the command pattern in combination with preconditions, the observer pattern, complex commands and analysis reports. An overview is given in Table 1.

4.2.1 Consistency by monitoring

This approach restricts the modeler to enter specifications that keep the meta-model consistent. Correctness is enforced in MERMAID by following this strategy: all rules for the manipulation of meta-model objects are implemented as preconditions to atomic commands (see Fig. 3). For example the addition of any meta-object type takes the following steps in the Create event handler: first, the meta-object is constructed using the input-parameters collected from the tool's user interface. Then one test class per meta-model object checks all necessary constraints: for example uniqueness of names, no loops in inheritance relationships, a specialized method should depend on an inherited method,... Finally if all tests succeed, the Add-command is executed and the meta-object is added to the repository; otherwise the failed test is reported to the user.

Table 1. Overview of consistency management techniques and their implementation

	Monitoring	Construction	Analysis	Completeness
Command pattern and preconditions	X			
Observer pattern		X		X
Complex command		X		X
Analysis report			X	X

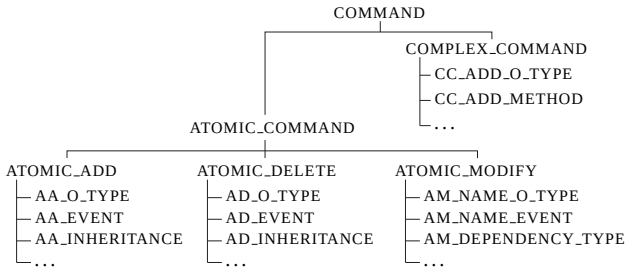

Figure 3. Elaboration of the Command pattern

4.2.2 Consistency by construction

In this second approach all required actions to maintain consistency are automatically inferred after the analyst adds a new specification. In MERMAID this is achieved by implementing the Observer pattern [10] and using complex commands as extension to the Command pattern as shown in Table 1.

The Observer pattern is used to keep the three views consistent. This can be seen as a *passive* form of consistency by construction: as the user changes a specification in one view, it is necessary to inform all other views about

the modification. In MERMAID the Command pattern has been integrated with the Observer pattern, which leads to a very modular structure: the right subclass of the command class acts as a subject that notifies all observers whenever the state of the subject changes, i.e. after executing a command. The abstract class ATOMIC_ADD ensures that all observers will be informed that a new object has been created after executing a command by means of a descendent command class.

The implementation of complex commands is the other, *active* way to ensure consistency: by grouping a set of atomic commands into one complex command we can enforce the completion of the entered specifications. For example the complex command of adding an object type creates besides the object type itself also the default finite state machine for that object type. The complex command (CC_ADD_O_TYPE) builds a list of atomic commands to create an object type (AA_O_TYPE), a finite state machine (AA_FSM_ROLE) and its initial, intermediate and final state (three times AA_STATE). In that way, the alphabet rule and the default lifecycle rule are obeyed and the tool itself can at any point in time rely on the existence of that state machine. The complex command to add a method (CC_ADD_METHOD) will add an appropriate transition (AA_TRANSITION) to the default finite state machine besides the method itself (AA_METHOD).

4.2.3 Consistency by analysis

As already stated in the introduction, inconsistencies in an information system do not always have to be seen as negative elements [9]. Despite the implementation of consistency by monitoring and consistency by construction approaches, one cannot avoid that some inconsistencies cannot be prevented by these approaches. In the first place, some inconsistencies can only be detected by an overall analysis of the repository. Secondly, as the user has the possibility to delete meta-model objects (like deleting a method, a transition, ...), a consistent specification can be made inconsistent by the deletion of a meta-model object. Thirdly, some constraints are only applicable to a final model and should therefore not be enforced on a model that is still under development. And finally, in MERMAID, the user has the possibility to turn off some of the consistency by construction options, mainly for educational purposes. As a result it is indispensable to implement some analysis tools to check specifications for inconsistencies.

Finite State Machine Analysis An example of the first type of inconsistencies is revealed by the analysis tool for finite state machines. As a result of the consistency by construction approach, the developer will initially receive a ready-made default finite state machine in which (s)he can add, modify or delete states and transitions to model the behavior. It is possible at any time to generate a report with consistency check results. The analysis tool checks the FSM for forward and backward inaccessible states, i.e. states that don't have a path to the initial state or a final state respectively. The finite state machine is also inspected for non-determinism: if one state has two or more transitions to different states for the same method, the finite state machine is non-deterministic. These inconsistencies cannot be detected when one transition is inserted, modified or deleted: they can only be detected by an overall analysis of the FSM. Inconsistencies can also occur because of the deletion of some meta-model object. Therefore, the FSM analysis tool also verifies the FSM against the alphabet rule: although this rule is initially satisfied when the default FSM is generated, it might not be satisfied any more after the deletion of a transition. In addition, if more than one FSM has been specified for one class, an analysis tools allows to calculate the parallel composition of all the state machines for one class, enabling in this way a semantic check by the user. This allows to check whether the composition is deadlock-free and whether all methods are still present in the combined behavioral description. These kinds of checks are similar to those presented in [21].

Object-Event Table Analysis In MERMAID the user has the possibility to turn off some of the consistency by construction options, mainly for pedagogical reasons. These possibilities affect the completeness and consistency of the object-event table. As an example, the user can choose not to automatically generate the required creating and ending methods for new object types, violating in this way the alphabet rule. The automatic propagation of methods can also be disabled resulting in the propagation rule and inheritance rule not being automatically fulfilled. In both cases, the user can manually complete the model. The OET analysis procedure creates and flags all the missing elements in the OET. The graphical user interface will show the missing elements in red and the validated elements in green. Upon leaving the analysis report, all computed items are deleted to put the specification back in its original state.

5. Comparison to Existing Work

Work on consistency management for UML can broadly be classified in two categories. The first category is the research that focuses on the definition of formal semantics for UML. A second category is research that focuses on the development of consistency checking tools, with or without prior translation of UML-specifications to a formal language.

A more rigorous definition of UML semantics has been a major concern for many years [13, 17, 20]. UML is a

typical committee language, which is very the complex as a result of the many compromises made to accommodate for all partner's concerns. Due to this complexity, a complete formal definition that encompasses all possible concepts of UML is a big challenge. Most research present a partial formalization of UML, focusing either on a single diagram [1, 8, 23] or on the relationship between two diagrams (e.g. the definition of state machine inheritance in relation to the generalization/specialization hierarchy [14], the integration of lifecycle model and interaction model [2, 4] or the integration of behavior and the notion of composition [3]). Still another approach aims at a more formal approach to the UML meta-model [17, 31]. Most of these efforts do however not explicitly define consistency rules or implement a consistency management tool.

More recently, tools have been proposed for checking the consistency of UML-specifications. As a commonly agreed set of constraints is still lacking, some research concentrates on the development of rule-based engines where rules can be defined in some proprietary language. In [28] a Constraint Checker is proposed based on an expert system. The tool uses a native language for rule formulation and checks UML specifications that have been converted to an XML-format. The rule language allows for the specification of actions that can generate recommendations for the modeler. In the future a tool-specific API should allow a tighter connection of the Constraint Checker with a Case-tool. Xlinkit [9] uses a combination of XML, XLink and XPath to check a set of UML documents. The expressivity of rules is limited by the expressivity of XPath and does not allow to specify actions. On the other hand, the tool offers the possibility to generate hyperlinks as a diagnostic instrument for the consistency status of a set of documents. Both tools follow the consistency by analysis principle: checking UML-specifications is a separate non-incremental static activity. The checking is done *after* the specification process and in a non-incremental way: each time the model is checked against the complete rule-base. In [32] a plug-in is presented for the Fujaba tool suite, which allows for incremental consistency checks and flexibility of rule-definition.

Other approaches rely on the transformation of UML specifications to some mathematical formalism. The consistency checking capabilities of such approaches is strongly determined by the complexity of the underlying mathematical formalisms. For most formalisms, it is already known which problems are decidable or undecidable and the complexity of algorithms for decidable problems are usually known as well. As an example, equivalence of Petri-Nets is known to be undecidable for the general class of Petri-Nets, whereas equivalence of Finite State Machines is decidable in polynomial time. For this reason, the formalism must be carefully chosen and restricting formalism to a subclass can ensure decidability in some cases. As an example, [30] relies on the transformation into Description Logic. As opposed to first order logic, where satisfyability is known to be undecidable, most Descriptio Logics have a decidable inference mechanism. This approach is in a prototype stage and uses the LOOM tools. In [7] UML models are translated to the Petri-net formalism for analysing the behavioral aspects and in [21], the underlying formalism is the failure divergence model of CSP. It is not clear to what extent these approaches suffer from the traceablity problem: to what extent can a reported inconsistency be traced back to the original UML model ?

The tool presented in this paper is to be classified in this second type of approach, characterized by a common formal model for all views. The graphical representation is a direct representation of the formal specifications: there is no separate translation step. Compared to the rule-based approach, our approach has the disadvantage that the rules are hard-coded in the tool. They are closely linked to the methodology used and cannot be changed dynamically. There is however flexibility in this sense that the user can turn some of the rules on or off.

In all the mentioned proposals there is a large variety in the range of constraints that can be checked. Some examples of rules are typical "referential" integrity constraints such as dangling references and missing definitions [30]. Process algebraic approaches are much stronger at behavioral analysis: they allow for the detection of deadlock and superfluous events and methods, see e.g. [21].

The MERMAID tool implements a large variety of constraints. The constraints mentioned in Sect. 3 come on top of referential integrity constraints. Dangling references and missing definitions (like a classless state diagram) are made impossible by a combination of referential integrity constraints implemented as pre-conditions for statement insertion and as cascading deletes for statement deletion. The tool will for example not allow to create a Finite State Machine without specifying a class as context for this FSM. On the other hand, the deletion of a class will invoke a garbage collector which will collect all the data referencing this class. This list of objects is shown to the user prior to confirmation of the delete-operation. These kinds of constraints can much more easily be implemented because of the restriction of UML to a limited set of diagrams and concepts.

With the exception of [32], most consistency approaches present a consistency by analysis strategy. MERMAID combines all three approaches. This has the major advantage of allowing an incremental approach. The monitoring approach checks only the statements that are inserted, modified or deleted. Statements that are automatically inserted by the construction approach are automatically consistent. As a result, the analysis approach is only used for those constraints that need an overall model analysis.

6. Conclusion

UML defines a set of nine different views on an information system. Each view (or model) is defined by a meta-model and a set of well-formedness rules. Achieving a single model approach with UML is rather difficult because of the lack of precise and formal semantics. Inter-view consistency rules are not explicitly identified, but only implicitly present because meta-classes appear in the meta-model of more than one diagramming technique. Although the semantics guide of UML [29] provides a start towards a more formal and unambiguous definition of UML, there is still a long way to go. Because of the remaining gaps in the formalization of UML, most case-tools that implement UML define de facto some form of consistency, but never implement all rules of the semantics guide completely.

Most existing approaches to consistency management follow the consistency by analysis principle. In this paper we have presented a tool that implements consistency by monitoring and consistency by construction strategies in addition to the more wide-spread approach of consistency by analysis. The use of the Command pattern allows to implement the consistency by monitoring approach and the use of complex commands and the Observer pattern allows for the realization of consistency by construction. The tool is complemented by a number of analysis reports that search for those inconsistencies that can only be detected by an overall analysis of the specifications. Approaches that use a separate rule-base, have the advantage of being more generic than the hard coding of consistency that is used in MERMAID. However, in comparison with consistency by analysis techniques, the proposed approach offers the advantage of the more user-friendly and incremental consistency by monitoring and construction. On the other hand, an additional verification on the XMI-output of the tool would allow to verify the specifications in the tool against other requirement documents.

In [26] we have demonstrated that consistency by construction is an important tool in ensuring the completeness of specifications. In addition the consistency by monitoring approach avoids inconsistencies and hence a lot of unnecessary rework. The automatic generation of specifications is also a means to avoid a “big bang” approach to quality, that is to say, an approach where quality is only checked at the end of the specification process, causing rework and delay.

Further research will expand the tool’s capabilities to more UML diagrams and concepts, offering gradually a more complete support for UML.

References

[1] Bourdeau R. H., Cheng B. H. C.: A formal semantics for object model diagrams, IEEE Transactions on Software Engineering, 21 (10), October 1995, pp. 799-821

[2] Bruel J. M., Lilius J., Moreira A., France R. B.: Defining Precise Semantics for UML, ECOOP 2000 Workshop Reaer, LNCS 1964, Springer 2000, pp.113-122.

[3] Brunet J.: An enhanced definition of Composition and its use for Abstration, in Proceedings of the International Conference on Object Oriented Information Systems, 9-11 September, Paris, 1998

[4] Cheung K. S., Chow K. O., Cheung T. Y.: Consistency analysis on lifecycle model and interaction model, in Proceedings of the International Conference on Object Oriented Information Systems, 9-11 September, Paris, 1998

[5] Dedene G., Snoeck M., Formal deadlock elimination elimination in an object oriented conceptual schema, Data and Knowledge Engineering, 15 (1-30), 1995

[6] Derrick J., Akehurst D., Boiten E., A framework for UML Consistency, in [13]

[7] Engles G., Heckel R., Kster J. M., Groenewegen L.: Consistency-preserving Model Evolution through Transformations, in J.M. Jzquel, H. Hussmann, S. Cook (eds.), UML 2002, The Unified Modeling Language, LNCS 2460 Springer Verlag, 2002, pp. 212-226

[8] Evans A., France R., Lano K., Rumpe B.: Developping the UML as a Formal Modelling Notation, in UML’98 Beyond the notation; International Workshop Mulhouse France, P-A. Muller, J; Bzivin (eds.), 1998

[9] Finkelstein A., Gabbay D., Hunter A., Kramer J. Nuseibeh B.: Inconsistency Handling in Multi-Perspective Specifications, IEEE TSE, 20(8): 569-578, 1994

[10] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995

[11] Harel David, Rumpe Bernhard, Modeling Languages: Syntax, Semantics and All That Stuff, Technical paper number MCS00-16, The Weizmann Institute of Science, Rehovot, Israel, 2000. avialable from [20]

[12] Hnatkowska B., Huzar Z., Kuzniarz L., Tuzinkiewicz L., A Sytematic approach to consistency within UML based software development process, in [13]

[13] Kuzniarz L., Reggio G., Sourrouille J. L., Huzar Z.: Workshop on Consistency Problems in UML-based software development, Workshop Materials, Research Report 2002:06, Blekinge Institute of Technology, Ronneby 2002, Workshop at the UML 2002 Conference, online at http://www.ipd.bth.se/consistencyUML/

[14] Le Grand: Specialisation of Object Lifecycles, in Proceedings of the International Conference on Object Oriented Information Systems, 9-11 September, Paris, 1998

[15] Lindland O.I., Sindre G., Sølvberg A.: Understanding Quality in Conceptual Modeling. IEEE Software, March 1994, pp. 42-49

[16] Model-driven Architecture, http://www.omg.org/mda

[17] Naumenko A., Wegman A.: A Metamodel for the Unifeid Modeling Language, in J.M. Jzquel, H. Hussmann, S. Cook (eds.), UML 2002, The Unified Modeling Language, LNCS 2460, Springer Verlag, 2002, pp. 2-17

[18] Nuseibeh B., Easterbrook S., Russo A.: Leveraging Inconsistency in Software Development, IEEE Computer, April 2000, pp. 24-29

[19] Paige R., Ostroff J.: The Single Model Principle. Journal of Object Technology, vol. 1, no. 5, November-December 2002, pp. 63-81. online available at http://www.jot.fm/issues/issue_2002_11/column6

[20] pUML, The precise UML group, http://www.cs.york.ac.uk/puml/

[21] Rash H. Wehrheim H., Consistency between UML classes and Associated State Machines, in [13]

[22] Rumpe B.: A note on Semantics (with an emphasis on UML), in Second ECOOP Workshop on Precise Behavioural Semantics, H. Kilov, B; Rumpe (eds.), Technische Universitt Mnchen, TUM-I9813, 1998

[23] Saksena M., France R. B., Larrondo-Petrie M. M.: A characterization of Aggregation, in Proceedings of the International Conference on Object Oriented Information Systems, 9-11 September, Paris, 1998

[24] Shaw M., Garlan D.: Software architecture: perspectives on an emerging discipline, Prentice Hall, 1996, 242 p.

[25] Snoeck M., Dedene G.: Existence Dependency: The key to semantic integrity between structural and behavioral aspects of object types. IEEE Transactions on Software Engineering, 24(24), 233-251.

[26] Snoeck M., Michiels C., Dedene G.: Consistency by construction: the case of MERODE. in Jeusfeld, M. A., Pastor, O., (Eds.) Conceptual Modeling for Novel Application Domains, ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Chicago, IL, USA, October 13, 2003, Proceedings, 2003 XVI, 410 p., Lecture Notes in Computer Science, Volume 2814, Springer Verlag 2003 pp.105-117.

[27] Snoeck M., Dedene G., Verhelst M., Depuydt A.M.: Object-oriented Enterprise Modelling with MERODE. Leuven, Leuven University Press. 1999

[28] Sourouille J.L., Caplat G., Checking UML Model Consistency, in [13]

[29] UML, OMG, http://www.omg.org/UML

[30] Van der Straeten R., Mens T., Simmonds J., Jonckers V.: Using Description Logic to Maintain Consistency between UML Models, in P. Stevens, J. Wittle, G. Booch, UML 2003, The Unified Modelling Language, LNCS 2863, Springer Verlag, 2003, pp.326-340

[31] Varro D., Pataricza A.: Metamodelling Mathematics: A Precise and Visual Framework for describing Semantic Domains of UML Models, in J.M. Jzquel, H. Hussmann, S. Cook (eds.), UML 2002, The Unified Modeling Language, LNCS 2460 Springer Verlag, 2002, pp. 18-33

[32] Wagner R., Giese H., Nickel U.A., A Plug-in for Fexible and Incremental Consistency Management, in Kuzniarz L., Huzar Z., Reggio G., Sourrouille J. L., Staron M.: Workshop on Consistency Problems in UML-based software development II, Workshop Materials, Research Report 2003:06, Blekinge Institute of Technology, Workshop at the UML 2003 Conference,

[33] Wirsing M., Knapp A., View Consistency in Software Development, In Martin Wirsing, Alexander Knapp, and Simonetta Balsamo, editors, Proc. 9th Int. Wsh. Monterey. Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02). Revised Papers, volume 2941 of Lect. Notes Comp. Sci., pages 341-357. Springer, Berlin, 2004