Consistency by construction: the case of MERODE

Monique Snoeck, Cindy Michiels, Guido Dedene

Department of Applied Economic Sciences, Katholieke Universteit Leuven,
Naamsestraat 69, 3000 Leuven, Belgium
monique.snoeck, cindy.michiels, guido.dedene@econ.kuleuven.ac.be

Abstract. Modeling languages such as UML offer a set of basic models to describe a software system from different views and at different levels of abstraction. Tools supporting an unrestricted usage of these UML models cannot guarantee the consistency between multiple models/views, due to the lack of a formal definition of the semantics of UML diagrams. A better alternative that does allow for automatic consistency checking is modeling according to the single model principle. This approach is based on the conception of a single model, for which different views are constructed, and with an automatic or semi-automatic generation or consistency checking among these views. Three basic approaches to consistency checking are consistency by analysis, consistency by monitoring and consistency by construction. In this paper we illustrate the consistency by construction approach by means of the conceptual domain modeling approach MERODE and its associated case-tool MERMAID. We also illustrate how consistency by construction improves the validity and completeness of the conceptual model.

1. The single model principle

The framework of Lindland, Sindre and Solvberg 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 [1]. 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[1]. In order to do automatic consistency checking, the model must be captured in a formal language.

Modeling languages such as UML offer a set of basic models to describe a software system from different views and at different levels of abstraction [2]. Examples of models included in UML are Use Cases for the functional requirements, class diagrams for the static view, interaction diagrams for the dynamic view, etc. Tools sup-

[1] As opposed to *external* correctness, meaning that a specification should meet the user requirements.

porting an unrestricted usage of these UML models, cannot guarantee the consistency between multiple models/views of the same system if these are constructed independently. The reason why automatic consistency checking cannot be supported is that UML lacks formal rules to enforce a consistent mapping between the models it defines. A better alternative that does allow for automatic consistency checking is modeling according to the single model principle [3]. This approach is based on the conception of a single model, for which different views are constructed, and with an automatic or semi-automatic generation or consistency checking among these views[2].

For the verification of view consistency three basic approaches can be distinguished. A first approach is *consistency by analysis*, meaning that an algorithm is used to detect all inconsistencies between two 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 errors and/or incompleteness in the various views. The verification can be done manually, but obviously building the algorithm into a case-tool will substantially facilitate the consistency checking procedure.

The second approach can be denoted as *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 is constantly consistent. Whereas the first approach puts the burden of correcting inconsistencies on the requirement engineer, the second approach avoids the input of inconsistencies. At the end of the specification process, the model must still be verified for completeness. The possible disadvantage of this approach is that a too stringent verification procedure will turn the input of specifications into a frustrating activity. The two approaches can be compared to two spelling and grammar checking strategies in word processing: the first checks spelling and grammar by running the spelling and grammar checker periodically, whereas the second approach is the equivalent of the option "check spelling and grammar as you type".

A third approach is *consistency by construction*, 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. The major advantage is however that the specifications are more or less constructed in an automated way: everything that can automatically be inferred is generated by the case-tool. This saves a lot of input effort. In addition, whereas the monitoring approach leads to a case-tool that generates error messages at every attempt to enter inconsistent specifications, the self-constructing approach avoids the input of inconsistent specifications by completing the entered specifications with their consistent consequences. The result is a much more user-friendly environment. Moreover, the automated generation of specifications offers the major advantage of improved completeness of the model.

In the remainder of the paper, we discuss the integration of the consistency by construction approach in MERMAID, a modeling tool based on the object-oriented analysis method MERODE. This methodology offers three basic views on a business

[2] Notice that in UML, each view is called a "model".

model –static, dynamic and interaction view– and is formalized in a set of rules managing all mappings between these views. Since the aim of the paper is to illustrate the modeling gains of a tool supporting the single model principle by consistency by construction, we kindly ask the reader to take the methodology "as is".

The paper is organized as follows. Section 2 introduces the three views that are supported in MERODE. Section 3 then briefly presents the consistency checking rules as they have been elaborated in [4][5] and discusses consistency by construction in MERMAID (due to space limitations, inheritance will not be discussed). Section 4 illustrates how consistency by construction improves the validity and completeness of the conceptual model. Finally section 5 presents some conclusions.

2. Overview of the Static, Dynamic and Interaction View

MERODE stands for Model-driven Existence dependency Relation, Object-oriented DEvelopment. It is a methodology for object-oriented enterprise modeling that has grown out of research on semantic modeling approaches, Jackson Systems Development [6] and object-oriented analysis.

The most distinguishing features of this methodology are its specific orientation to domain modeling, the use of Existence Dependency to model the static aspects of the domain model, and the event driven approach to behavior modeling. Relevant concepts will be explained in subsequent sections. By means of an example will be illustrated how a specification can be self-completing to a certain extent and how this automated consistency by construction contributes to the validity and completeness of specifications. A MERODE model consists of three subviews:

- an existence dependency graph (EDG) that organizes enterprise object types according to existence dependency and inheritance,
- an object-event table (OET), which identifies business event types and relates those to the enterprise object types,
- a behavioral model, consisting of one finite state machine (FSM) per enterprise object type.

The semantics of the EDG, the OET and the FSMs have been defined by means of process algebra and view consistency has been defined at the same time [4][5]. As a result, a set of consistency checking rules is available for this method, which also provide for some basic completeness check. Fig.1 gives an overview of the views and the rules.

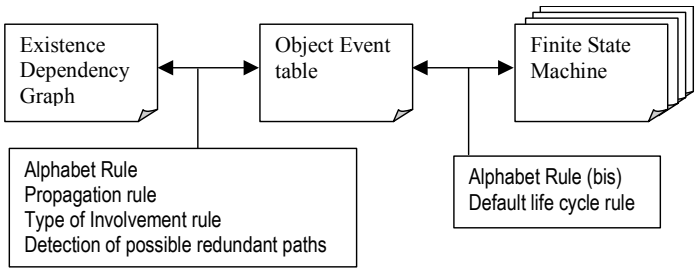

Fig. 1. Views and consistency checking rules in MERODE

2.1 The Existence Dependency Graph

Let us consider the UML class diagram in Fig. 2. It represents a situation where customers can place zero to many orders for projects. Each project is ordered by exactly one customer. Employees work on projects: each employee works on exactly one project at a time and each project has zero to many employees working on it.

Fig. 2. Project management

Although the two associations look identical in their graphical representation, there is some substantial difference in the semantics of each association. Indeed, every employee works on one project *at a time,* but *over time* employees can work on several projects *consecutively*. In other words, the association "works for" is modifiable. The "orders" association however, is not modifiable: a project is ordered by one customer, but this customer remains the same over time. Consequently the diagram in Fig. 2 can be considered to be semantically incomplete: some relevant statements about the domain have not been expressed. Therefore, in MERODE, it is required to transform a class diagram into an existence dependency graph (EDG). In such graph, all object types are only related through associations that express existence dependency. According to the formal definitions in MERODE, 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. D is called the *dependent class* and is existence dependent of M, called the *master class*. A more informal way of defining existence dependency is as follows: if each object of a class D always refers to minimum one, maximum one and always the same occurrence of class M, then D is existence dependent of M. Notice that existence dependency is equivalent to the notion of weak entity as defined by Chen [7][4]. To avoid confusion with a standard UML class diagram, MERODE uses a proprietary notation with dots and arrows to define cardinality of the existence dependency relationship. This cardinality defines how many occurrences of the dependent object type can be dependent of one master object at one point in time. As the cardinality of the master class is always exactly one (every dependent is associated to exactly one master), only the cardinality for the dependent needs to be specified. An arrowhead means that the master can have several dependents simultaneously whereas a straight line limits the maximum cardinality to one. A white dot means that having a dependent is optional for the master, whereas a black dot imposes a minimum constraint of one (the master has at least one dependent at any time).

The transformation of the class diagram of Fig. 2 results in the EDG of Fig. 3. The "orders" association expresses existence dependency: each project can only exist within the context of a customer and refers to exactly one and always the same customer for the whole duration of its life. A customer on the contrary can exist on its own. He needs not to have a project in order to exist (optionality indicated by the white dot) and he can have many ongoing projects (arrowhead). The "works for" relationship does not represent existence dependency. An employee can exist outside of the context of a project and a project can exist outside of the context of an employee. When an association does not express existence dependency, the associa-

When an association does not express existence dependency, the association is turned into an object type that is existence dependent of all the object types participating in the association. In this case this means that the "works for" association is turned into an object type ASSIGNMENT, which is existence dependent of PROJECT and EMPLOYEE. MERODE calls this type of intermediate class a "contract" class: it models what can happen during the period of time that a project and an employee are related to each other. Since a project can have zero to many employees, each project has zero to many assignments (white dot, arrow). And as each employee is assigned to exactly one project at a time, each employee has exactly one assignment at a time (black dot, straight line).

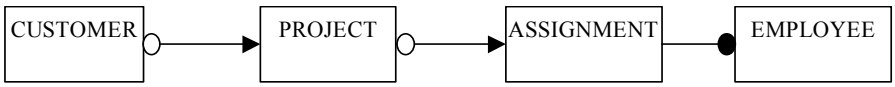

Fig. 3. Existence dependency graph for the project management example.

2.2. The Object-Event Table

In the case of object-oriented conceptual modeling, domain requirements will be formulated in terms of business or enterprise object types, associations between these object types and the behavior of business object types. The definition of desired object behavior is an essential part in the specification process. On the one hand, we have to consider the behavior of individual objects. This type of behavior will be specified as methods and statecharts for object classes. On the other hand, objects have to collaborate and interact. Typical techniques for modeling object interaction aspects are interaction diagrams or sequence charts, and collaboration diagrams.

In most object-oriented approaches events are considered as subordinate to objects, because they only serve as a trigger for an object's method. The object interactions themselves are modeled by means of sequence and/or collaboration diagrams.

In contrast, MERODE follows an event-driven approach that raises events to the same level of importance as objects, and recognizes them as a fundamental part of the structure of experience [8]. A business event is now defined as an atomic unit of action that represents something that happens in the real world, such as the creation of a new customer, an order placement, etc. The business events reflect how domain objects come into existence (the creating events), how domain objects are modified (the modifying events), and how they disappear from the universe of discourse (the ending events). Object interaction can now be modeled by defining which objects are concurrently involved in which events. Object-event participations are denoted by means of an object-event table (OET). When an object participates in an event, it implements a method that defines the effect of the event on the object. On occurrence of the event all corresponding methods in the participating objects are executed in parallel. Thus, instead of modeling a complex sequence of method invocations, it is now assumed that all methods are concurrently executed. The OET for the project management example is given in Table 1. The rules that govern the construction of this table are described in the next section.

2.3 The finite state machines

Finally, the life cycle of every enterprise object class is modeled by means of a finite state machine (FSM). The events of the object-event table are used as triggers for the transitions in the finite state machine. As an example, Fig. 4 shows the FSM for EMPLOYEE. Similarly, a FSM can be defined for the classes PROJECT, ASSIGNMENT and CUSTOMER.

Table 1. Object-event table for project management.

	Customer	Project	Employee	Assignment
cr_customer	C			
mod_customer	M			
end_customer	E			
cr_project	M	C		
mod_project	M	M		
end_project	M	E		
cr_employee			C	
mod_employee			M	
end_employee			E	
assign	M	M	M	C
remove	M	M	M	E

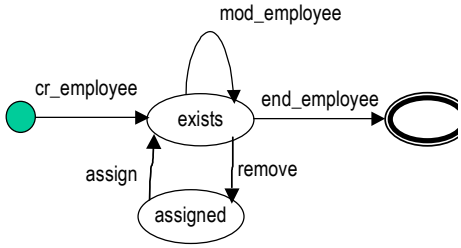

Fig. 4. Finite state machine for Employee

3. Consistency by construction

The construction of the OET is governed by a number of rules that ensure the consistency of the OET with the EDG. An algorithmic approach to consistency checking would verify the consistency *after* entering the specification. In this section we illustrate how many of the consistency rules allow to automatically generate some parts of the requirements, preventing in this way inconsistencies and incompleteness.

3.1 Alphabet Rule

The alphabet of an object class is defined as the set of all event types that are marked for this object type in the OET. The Alphabet Rule states that each event can have only one effect on objects of a class: the event either creates, modifies or deletes objects. In addition, the rule states that each object class needs at least one event to create occurrences and one event to destroy occurrences in this class.

Rather than verifying post factum whether there is at least one creating and one ending event for each enterprise object type, the case-tool will automatically generate two business events when an object type is added to the EDG. The default names are the name of the object type preceded by "cr_" and "end_", but as shown in Fig. 5, the user can overwrite the names and decide not to generate one or both event types. Simultaneously, the OET is completed accordingly: a column is added for the object type, two rows are added for the event types and the participations are marked (see Fig 6.).

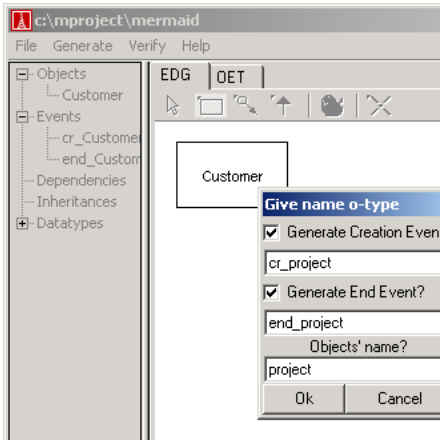

Fig. 5. Existence dependency graph

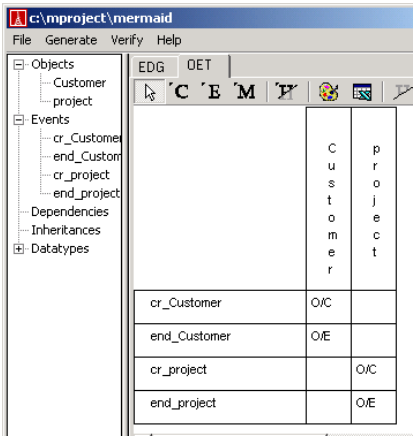

Fig. 6. Object-event table

3.2 Propagation Rule and Type of Involvement Rule

A second rule in the construction of the OET is the *propagation rule*. The propagation rule states that when an object type D is existence dependent of an object type M, the latter is by default also involved in all event types D is involved in. This means that if an involvement is marked for an event type in the column of a dependent object type D, it must also be marked in the column of the master object type M.

In addition, the *type of involvement rule* states that since an existence dependent object type cannot start to exist before its master, a creating event type for a dependent class is a creating or a modifying event type for the master class. A modifying event type for a dependent class is also a modifying event type for its master class. And finally, since a dependent cannot outlive its master, an ending event type for a dependent is an ending or modifying event type for its master. To discern the participations the master acquired from its dependents through the propagation rule from the event type participations that are proprietary to the master class, the former are preceded by a 'A/' (from **A**cquired) and the latter by an 'O/' (from **O**wned).

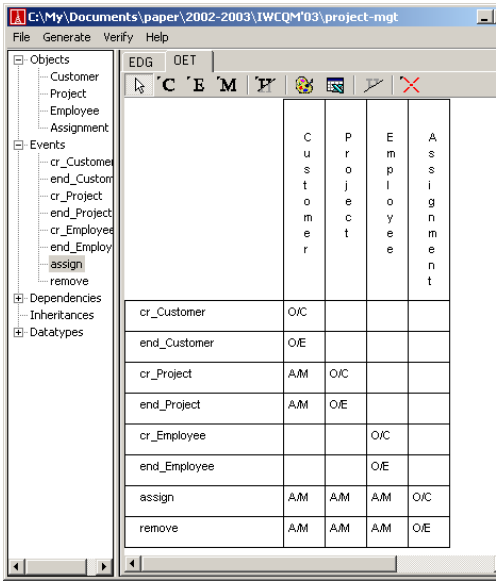

Fig. 7. OET with propagated object-event participations

Performing and verifying the propagation by hand is a time consuming task, especially for larger projects. A case-tool however, can easily generate all the propagated participations. For the project man-

agement example, the resulting OET after entering the four object types and the existence dependency relations is shown in Fig. 7.

The OET can be modified independently from the EDG, but also in this case, consistency is automatically enforced whenever possible. Adding an object type in the OET will add the object type in the EDG as well, although it will not be related to other object types already in the EDG.

Events can be added in the OET and for these events we can add owned methods, which will be automatically propagated. Acquired methods cannot be added or removed. The type of involvement can be modified, provided it follows the type of involvement rule.

3.3 Detection of Possible Redundant Paths

Joining paths in the EDG occur when a master can be reached from a dependent by following two different existence dependency paths transitively from dependent to master. Assume that the project management example is extended with invoicing as in Fig. 8. During his/her assignment to a project, each employee can register the hours performed for the project. This time registration is included on an invoice at the end of the month as an invoice line.

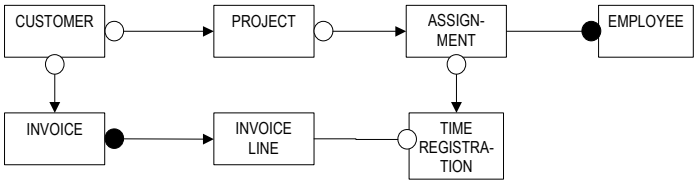

Fig. 8. Extended EDG for Project Management

Going along the existence dependency relations from dependent to master, the object type CUSTOMER can be reached in two ways from the class INVOICE LINE:

INVOICE LINE → INVOICE → CUSTOMER

and

INVOICE LINE→ TIME REGISTERATION → ASSIGNMENT → PROJECT → CUSTOMER

Applying the propagation rule in the OET automatically identifies this kind of path joins: path joins lead to multiple propagations in the OET. In the ordering example the object type CUSTOMER will acquire the event types from invoice line two times, once through each path (see Table 2). Identifying path joins is important since one must answer the question whether one or two customers are involved in an invoice line. In other words: is the customer for whom the work was done (that is to say, the customer connected to the project connected to the invoice line via assignment and time-registration) the same person as the one whom we send the invoice to? If this is the case, the double participation is replaced by a single participation and a constraint (an invariant) is added in the class INVOICE LINE:

self.INVOICE.CUSTOMER
= self.TIME_REGISTRATION.ASSIGNMENT.PROJECT.CUSTOMER

3.4 Alphabet rule and Default lifecycle rule

The alphabet rule also states that the FSM that defines the behavior of an object type P must contain all and only the event types for which there is a 'C', 'M' or 'E' in the column of P in the OET. In addition, the sequence constraints imposed by the FSM must not violate the default lifecycle of create, modify, end. Hence, according to these rules, a default FSM can be generated for each object type. This FSM can be further refined by adding new events and states.

Fig. 9 shows the FSM that can automatically be derived from the OET for TIME_REGISTRATION. This FSM can be further refined, for example to ensure that a time registration cannot be modified once it has been invoiced (as in Fig. 10). The case-tool ensures at any time that a creating event is only used for a transition departing from the initial state, that a modifying event is only associated to transitions between intermediate states and that an ending event is only associated with transitions terminating in a final state.

Table 2. Object-event table for the extended project management example

	CUSTOMER	PROJECT	EMPLOYEE	ASSIGN-MENT	TIME REGIS-TRATION	INVOICE	INVOICE LINE
cr_customer	O/C						
end_customer	O/E						
cr_project	A/M	O/C					
end_project	A/M	O/E					
cr_employee			O/C				
end_employee			O/E				
assign	A/M	A/M	A/M	O/C			
remove	A/M	A/M	A/M	O/E			
register	A/M	A/M	A/M	A/M	O/C		
modify_registration	A/M	A/M	A/M	A/M	O/M		
end_registration	A/M	A/M	A/M	A/M	O/E		
create_invoice	A/M					O/C	
pay_invoice	A/M					O/M	
end_invoice	A/M					O/E	
put_TR_on_invoice	A/M, A/M	A/M	A/M	A/M	A/M	A/M	O/C
modify_invoice_line	A/M, A/M	A/M	A/M	A/M	A/M	A/M	O/M
end_invoice_line	A/M, A/M	A/M	A/M	A/M	A/M	A/M	O/E

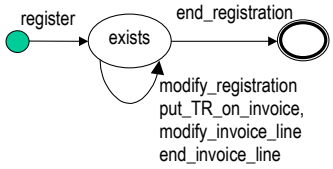

Fig. 9. Default FSM for TIME_REGISTRATION

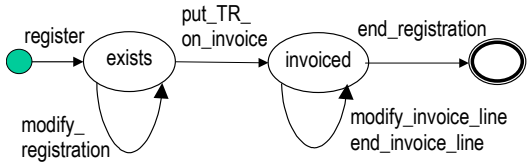

Fig. 10. Modified FSM for TIME_REGISIRATION

4. Completeness

Traditionally, modeling is viewed as a mapping of an area or part of the real world into a model [1][9]. In this view, 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.

When checking the completeness of a model, the user requirements are the reference point. Hence, user signoff is often considered to be a de facto measurement of completeness [10]. Unfortunately, users often don't even understand data models, let alone object-oriented conceptual models. It is therefore impossible to check completeness of a model by having it checked by users alone. In this respect, the automatic generation of those parts of the specifications that can be inferred from the already existing specifications will simplify the checking for completeness of the specifications. What can be inferred is however tightly connected to the semantics of the techniques for conceptual modeling. As an example, let us reconsider the class diagram of Fig. 2: it is semantically correct but incomplete as some relevant constraints were not identified nor explicitly incorporated into the model. It is certainly possible to add a note or a stereotype to express the differences between the two associations or else to express the difference in the behavioral model. The important point is however that the diagramming technique does not "enforce" the requirements engineer to think of the difference: it does not help in discovering the incompleteness in the model. By transforming this graph into an EDG, the incompleteness is resolved, resulting in a model that is semantically more complete. In the project management example, the transformation of the class diagram to an EDG leads to the creation of the object type ASSIGNMENT for the project management example. Subsequently, the alphabet rule requires the definition of a creating and an ending event type for the object type ASSIGNMENT, namely *assign* and *remove*. These event types allow specifying under what conditions it is allowed to assign and remove an employee to/from a project. The automatic generation of these two events helps in achieving the completeness of the model. Nothing in the original UML will point the requirements engineer to consider modeling these events.

In [4][5] the propagation rule is motivated as follows. Since and existence dependent object cannot exist outside the life of its master, anything that happens to the dependent also affects the master, at least indirectly. By notifying the master of the occurrence of the events on its dependents, the master class is able to do some accounting (e.g. in EMPLOYEE counting the number of projects an employee has ever worked on), or to enforce some constraints (e.g. PROJECT can set as precondition for the assign event that the state of the project should not be 'closed'). Again, the propagation rule illustrates how the automatic generation of object-event participations makes the specifications more complete: by propagating event type participations, all possible places for constraint definitions and information gathering are identified. In this way, the requirements engineer is invited to consider all these elements for the inclusion of potential business rules. In the end, when all requirements have been collected, some of the marked cells might have no constraint or method body associated with them. Those participations can easily be removed before implementation. Again, the rules of MERODE improve the self-completing character of requirements.

Finally, the OET provides an automatic mechanism for identifying path joins, which in turn leads to the identification of relevant constraints in the domain.

5. Discussion

The key factor of the single model principle is the verification of consistency between the different views of a model. In [3], Paige and Ostroff illustrate how BON/Eiffel follows the single model-principle and how the Single Model Principle can be applied to UML/Java by using profiles. In order to achieve a single model approach, they strongly restrict the types of UML diagrams used: only class diagrams and collaboration diagrams are included in the deliverables of the approach. Paige and Ostroff also identify two types of dependencies between the deliverables: an automatic construction dependency where a tool generates one deliverable from another and guarantees semantic consistency and an algorithmic consistency checking dependency where an algorithm is used to detect all inconsistencies between two deliverables and a report is thereafter generated for the developers.

MERODE also strongly restricts the type of diagrams used in order to meet the single model principle. In addition the EDG takes an unusual approach to data modeling, but as explained in [4][5], it is exactly existence dependency that is the key to the semantic consistency checking.

Achieving a single model approach with UML is rather difficult because of the lack of precise and formal semantics. The need for formal underpinning of UML has long been recognized and significant advances have been made [11], [12], [13], [14]. Many of these efforts are however limited to the isolated definition of a single modeling notation [15], [13], [16], [17]. Advances have been made towards the integration of different UML views [18]. Examples of such integration efforts are the definition of state machine inheritance in relation to the generalization/specialization hierarchy [19], [20], the integration of life-cycle model and interaction model [21] [18] or the integration of behavior and the notion of composition [16].

In this paper we have illustrated how the MERODE case-tool addresses the consistency checking required to achieve the single model approach. In fact, the MERODE case-tool uses a mix of automated construction (consistency by construction) and algorithmic consistency checking (consistency by analysis). Indeed, since the requirements engineer can further modify the diagrams, the automatic construction must be complemented by an algorithmic verification for those parts of the diagrams that were not constructed automatically. As an example, the MERODE case-tool provides an algorithm for checking FSMs for unreachable states. However, because a large part of the specifications were generated automatically, the number of remaining inconsistencies that have to be detected by algorithmic verification is much smaller than if the three views were built in an independent manner. 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.

An additional benefit of the automatic construction of specification is that it helps to improve the completeness of the specifications.

Since its creation, the MERMAID case-tool has proved its usefulness in several real-life projects, the largest of which counts over 44 enterprise objects and 134 business events. Since MERODE only covers the domain modeling part of a project, the tool has been provided with an XMI [22] interface. This allows exporting the specifications to other case-tools, e.g. those that support all types of UML diagrams.

References

[1] Lindland, O.I., Sindre, Guttom, Sølvberg, Arne, Understanding Quality in Conceptual Modeling, IEEE Software, March 1994, pp. 42-49

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

[3] Richard Paige, Jonathan Ostroff: "The Single Model Principle", in 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

[4] 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.

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

[6] Jackson M. Cameron J., System Development, Prentice Hall (1983).

[7] Chen, P.P., The Entity Relationship Approach to logical Database Design, *QED information sciences,* Wellesley (Mass.),1977

[8] Cook, S., Daniels, J.: Designing object systems: object-oriented modelling with Syntropy. Prentice Hall (1994)

[9] Schuette, R., Rotthowe, T., The Guidelines of Modeling - An Approach to Enhance the Quality in Information Models, In Tok Wang Ling, Sudha Ram,Mong Li Lee (eds), Conceptual Modeling - ER'98, 17th Interntional Conference on Conceptual Modeling, Singapore, LNCS 1507, Springer.

[10] Moody D.L., Shanks, G.G., Darke, P., Improving the Quality of Entity Relationship Models - Experience in Research and Practice, In Tok Wang Ling, Sudha Ram,Mong Li Lee (eds), Conceptual Modeling - ER'98, 17th Interntional Conference on Conceptual Modeling, Singapore, LNCS 1507, Springer.

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

[12] 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/

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

[14] Rumpe, A note on Semantics (with an emphasis on UML), in *Second ECOOP Workshop on Precise Behavioural Semantics*, H. Kilov, B; Rumpe (eds.), Technische Universität München, TUM-I9813, 1998

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

[16] J. Brunet, 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

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

[18] 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.

[19] M. Snoeck, G. Dedene, Generalisation/Specilisation and Role in object-oriented conceptual modelling, *Data and Knowledge Engineering*, 19(2), 1996

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

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

[22] OMG, XML Metadata Interchange, http://www.omg.org/technology/documents/formal/xmi.htm