Conceptual Modeling of Structure and Behavior with UML – The Top Level Object-Oriented Framework (TLOOF) Approach

Iris Reinhartz-Berger

University of Haifa
Carmel Mountain, Haifa 31905, Israel
iris@mis.hevra.haifa.ac.il

Abstract. In the last decade UML has emerged as the standard object-oriented conceptual modeling language. Since UML is a combination of previous languages, such as OOSE, OMT, Statecharts, etc., the creation of multi-views within UML was unavoidable. These views, which represent different aspects of system structure and behavior, overlap, raising consistency and integration problems. Moreover, the object-oriented nature of UML set the ground for several behavioral views in UML, each of which is a different alternative for representing behavior. In this paper I suggest a Top-Level Object-Oriented Framework (TLOOF) for UML models. This framework, which serves as the glue of use case, class, and interaction diagrams, enables changing the refinement level of a model without loosing the comprehension of the system as a whole and without creating contradictions among the mentioned structural and behavioral views. Furthermore, the suggested framework does not add new classifiers to the UML metamodel, hence, does not complicate UML.

1 Introduction

Conceptual modeling is fundamental to any domain where one has to cope with complex real world systems. The real world exhibits two separate aspects: structure (objects, nouns, etc.) and behavior (operations, verbs, etc.). Although being different aspects, structure and behavior are highly intertwined in the real world: operations get input objects, operation might change structures, sentences include both nouns and verbs, and so on. In spite of the differences between these two aspects in the real world, in the last few decades most of the modeling and programming languages are object-oriented, encapsulating behavior (operations) in structure (objects). The most popular, de-facto standard modeling language is Unified Modeling Language (UML) [14], which is used for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. UML defines ten types of diagrams, which are divided into three categories. Four diagram types represent structure and include the class, object, component, and deployment diagrams. Five diagram types, which represent different aspects of dynamic behavior, include use case, sequence, activity, collaboration (called

communication diagrams in UML 2.0), and Statechart diagrams. Finally, package diagrams represent ways to organize and manage application modules. UML 2.0 adds two more diagram types: timing diagrams for exploring the behaviors of one or more objects throughout a given period of time and composite structure diagrams for exploring run-time instances collaborating over communication links. A system modeled by UML consists of several different, but related and even overlapping, diagrams of various types.

The popularity UML gained and the standardization efforts of its creators made UML a common modeling language which is used in the various steps of system development, including during the requirement analysis, design, and testing phases. Many automatic code generators have been developed for generating code from UML models to various (especially object-oriented) programming languages. These usages of UML models require that they will be formal, complete, unambiguous, and consistent, in order to get qualitative systems.

Although UML provides a convenient, standard mechanism for software engineers to represent high-level system designs as well as low-level implementation details [19], several drawbacks prevent UML from being largely used in the industry. The main drawbacks are the fragmentation of UML views and the absence of solid glue between them, which arise syntactic and semantic *consistency problems*. UML syntactic rules relate to well-formedness of expressions, consistency of identifiers with their declarations, etc. Such rules, which are expressed in the UML metamodel, can be checked by diagram editors or CASE tools. Semantic consistency, on the other hand, is concerned with the compatibility of the meaning of the different views. Engels et. al. pointed two types of semantic consistency: horizontal and vertical [7]. *Horizontal consistency* refers to rules that should be preserved when traveling between different (overlapping) viewpoints of the same system, while *vertical consistency* concerns with rules that should be preserved during the different development stages.

The consistency problems of UML are also associated with *integration problem*s. The different UML views represent a single system. Humans engage in the development process, such as clients, users, designers, and implementers, should comprehend the system as a whole, complete unit. Moreover, automatic tools, for example code generators, should be able to generate a consistent, qualitative implementation from a UML multiple view model.

Several solutions for UML consistency and integration problems have been proposed over the years (e.g., [2], [4], [6], [12]). Most of them suggested using a formal language in addition to UML or running translation, verification, or testing algorithms on existing UML models. In this paper, I suggest a Top-Level Object-Oriented Framework (TLOOF) for creating complete, coherent UML models which capture both system structure and behavior. This approach enables explicit bindings between UML use case, class, and interaction diagrams, thereby supporting incremental development of consistent and integral UML models. A set of consistency rules between this framework and UML views is defined and exemplified.

The structure of the rest of the paper is as follows. Section 2 reviews and discusses the main consistency and integration problems of UML and some of their solutions. Section 3 presents the proposed framework, exemplifying it on a simple ordering

system. Section 4 defines consistency rules between the TLOOF framework and the other UML diagram types. Finally, Section 5 summarizes and discusses the benefits and shortcomings of the suggested solution and refers to future research plan.

2 Literature Review: Consistency and Integrity of UML Models

2.1 UML Consistency and Integration Problems

The need to model and design complex systems, which involve structural, behavioral, functional, and architectural aspects, introduced the notion of a *view*. Each (graphical or textual) view presents a different perspective of the system being developed. The actual views and the way in which system aspects are projected onto individual views are method- or language- dependant [9]. Although the usage of multiple views has great benefits in focusing on a specific aspect of the modeled system and in preserving the views in a reasonable size, it also raises consistency and integration problems.

As noted, Engels et. al. [7] divided UML consistency problems into horizontal and vertical ones. Horizontal consistency problems (also known as inter-model consistency problems) refer to contradictions that might exist due to the fact that the various views model the same system and the information resides at them overlaps. An example of a constraint related to horizontal consistency is: "Each Statechart must correspond to a state dependent class on a class diagram" [9]. The vertical consistency problems refer to inconsistencies or contradictions that exist when applying UML to the different development stages (due to the different abstraction levels of these stages). An example for this type of constraints is: "The information needed for implementing a use case must be described in a class diagram" [9]. While usually the data needed for checking horizontal consistency is explicitly modeled in the UML views, some of the information needed for verifying vertical consistency is implicit or expressed informally.

Another problem that exists due to the use of multiple UML views is misunderstanding of the system as a whole (i.e., *integration problems*). Using their framework for evaluating system analysis and design methods, Tun and Bielkowicz [20] claim that UML views (diagram types) are fragmented and there is little glue between them. Moreover, they assert that without rigorous crosschecking between the views, it would be hard to have confidence that the system would possess essential quality characteristics such as completeness, correctness, and consistency. Two experiments which compared a single-view methodology, Object-Process Methodology (OPM) [5], to multi-view modeling languages, Object Modeling Technique (OMT), the predecessor of UML, and UML ([16] and [18], respectively) showed that the single view of OPM is more effective than the multiple view modeling language in generating a better system specification. Most of the errors in the multiple view models resulted from the need to maintain consistency among the different view types and to gather information that is scattered across the views.

The consistency and integration problems of UML are also influenced from the existence of several behavioral views in UML, some of which represent specific scenarios rather than complete behavioral patterns. Uchitel et. al. [21] proposed an algorithm for synthesizing behavioral models from UML scenarios. Their algorithm translates a scenario specification to a Finite Sequential Processes (FSP) specification, which is then used for building a composite behavior model in the form of a labeled transition system (LTS). Several studies checked if there is any preference between UML behavioral views. Otero and Dolado [15], for example, compared the semantic comprehension of sequence, collaboration, and state diagrams. The comparison was in terms of the total time to complete tasks and their scores. They found that the comprehension of behavioral models in object-oriented designs depends on the complexity of the system. However, using sequence diagrams is the most comprehensible way to represent the system behavior. Hahn and Kim [11] conducted an experiment to check the effects of diagrammatic representation on the cognitive integration process of systems analysis and design. The researchers checked the comprehension of process components represented in sequence, collaboration, activity, and activity flow[1] diagrams. The results showed that decomposition of process components (which exists in sequence and collaboration diagrams) had a positive effect on both the analysis and design activities, while layout organization had a positive effect only on the design performance.

2.2 Solutions for UML Consistency and Integration Problems

Several solutions have been proposed for UML consistency and integration problems. These solutions can be divided into translation and verification approaches.

Translation approaches translate multi-view models into more formal languages of model checkers. The model checker tool is then deployed to analyze the given model for inconsistencies. Bowman et. al. [2], for example, use LOTOS in order to present a formal framework for checking consistency among various viewpoints in Open Distributed Processing (ODP). They define consistency between specifications X_1, X_2, …, X_n as the existence of a physical implementation which is a realization of all X_1, X_2, …, X_n. Furthermore, they classify consistency classes, such as binary consistency, complete consistency, balanced consistency, and inter language consistency, and express their characteristics using LOTOS. Rasch and Wehrheim [17] use Object-Z in order to give a precise semantics to UML class and Statechart diagrams and to check for consistencies between these views.

Mens et. al. [12] suggest restricting to description logic in order to specify and detect inconsistencies between UML models. They claim that the use of description logic is especially relevant since it contains five reasoning tasks that can be directly used to achieve subsumption, instance checking, relation checking, concept consistency, and knowledge base consistency.

[1] Activity flow diagrams are similar to activity diagrams, except that the activities are not arranged within swimlanes.

Große-Rhode [10] suggests a semantic approach for the integration of views. This approach, which is applied to the structural and behavioral views of UML, is based on transition systems, algebraic specifications, and transformation rules.

Engels et. al. [7] present a general methodology to deal with consistency problems in UML behavioral views. According to this methodology, relevant aspects of the models are mapped to a semantic domain in which precise consistency tests can be formulated.

Baresi and Pezze [1] suggest transforming fragments of UML models into high-level Petri nets that serve as a formal semantic domain. This way, UML behavioral views can be simulated and analyzed.

Verification approaches present testing or validation algorithms which check inconsistencies and contradictions between various views. Chiorean et. al. [4] use an OCL-based framework in order to ensure consistency among UML views. All the consistency rules are defined at the metamodel level, supporting their reuse for any specific user model.

Bodeveix et. al. [3] implemented a tool for checking the coherence between the different UML views. This tool is based on an OCL interpreter and a set of OCL expressions over the UML metamodel. Furthermore, OCL is extended to support temporal constraints over the behavioral views of UML.

Engels et. al. [6] propose dynamic metamodeling (DMM) as a notation for defining consistency conditions. DMM extends the metamodeling idea by introducing metaoperations for the metamodel classes. These operations encapsulate the dynamic semantics of the classes. A DMM-based testing environment, which consists of a test driver, a test controller, and DMM interpreters, was developed.

Based on a classification of consistency constraints that occur in and between specifications at various stages of the lifecycle, Nentwich et. al. [13] identify a set of requirements that consistency management mechanisms have to address in order to provide proper support. Examples of these requirements are flexibility in constraint application, a tolerant approach to consistency, support for distributed documents, etc. Using a lightweight for consistency checking framework that leverages standard Internet technologies, the researches address the consistency problems without requiring tight integration, complex translation of specifications, or bulky tools.

The mentioned translation approaches require definitions of translation rules from UML models to semantic, formal languages and definitions of consistency rules in yet other formal languages. This is usually done in two separate supporting tools: translation generators and model checkers, which together with UML-based CASE tools perform the environment in which the translation approaches exist. Moreover, after detecting inconsistencies a backward process should be applied, translating the locations where inconsistencies were found back to the UML models in order to enable the developers to fix the inconsistencies. The verification approaches require in addition sophisticated environments which include test drivers, interpreters, controllers, etc. Moreover, as noted, some of the consistency rules are not explicitly expressed in UML models, demanding semantic interpretation of the UML models and understanding the intentions of their developers.

While both the translation and verification approaches run one time algorithms for checking UML models after their development processes have been completed, I suggest verifying the legibility of the models during the development process. The

suggested approach requires defining a Top-Level Object-Oriented Framework, abbreviated as TLOOF, which glues the different views of a system under development and represents their relationships explicitly. The developers will be aware of existing inconsistencies at any specific time of the development process, thereby being able to correct the models as early as possible. Detecting inconsistencies in early development phases contributes to shortening the system's delivery time ("time-to-market").

3 The Top-Level Object-Oriented Framework Approach

The Top Level Object-Oriented Framework (TLOOF) approach introduces a TLOOF diagram which is actually an extension of a use case diagram. In addition to the actors and use cases which exist in regular use case diagrams, a TLOOF diagram includes collaborations and realization relations, both are already part of the UML vocabulary. *Collaborations* provide a way to group chunks of interaction behavior [8]. In other words, collaborations can be viewed as system processes that might have several possible scenarios, each of which should be described in a different interaction diagram. *Realizations* specify relationships between specification model elements and model elements that implement them. In particular they link use cases to collaborations. Collaborations are symbolized in UML as dashed ellipses, while realizations are denoted by dashed lines ending with triangles. Figure 1, for example, is a TLOOF diagram of an ordering system. This system requires that a customer will be able to find a product and order it. During the requirement analysis stage, three specification model elements are established: the actor **Customer** and the use cases **Product Finding** and **Product Ordering**. A more detailed specification could be written, dividing **Product Ordering** into searching, reserving, paying, and supplying, but this type of specification is not needed at the requirement level.

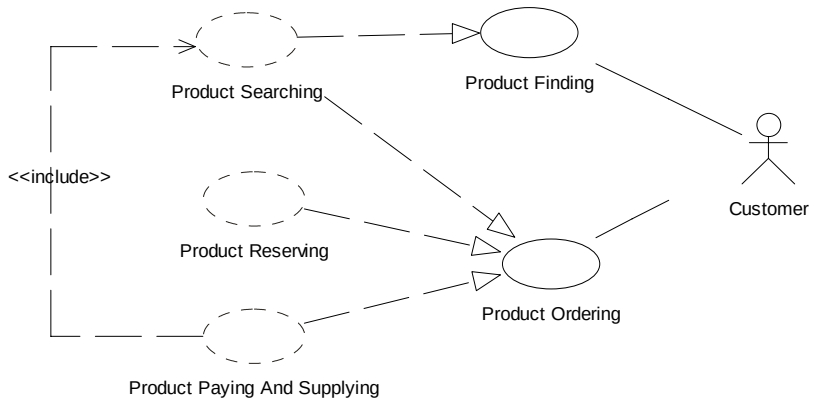


Figure 1. The TLOOF diagram of an ordering system

While designing the system, the developers find out that they have to implement three main processes: **Product Searching**, **Product Reserving**, and **Product Paying**

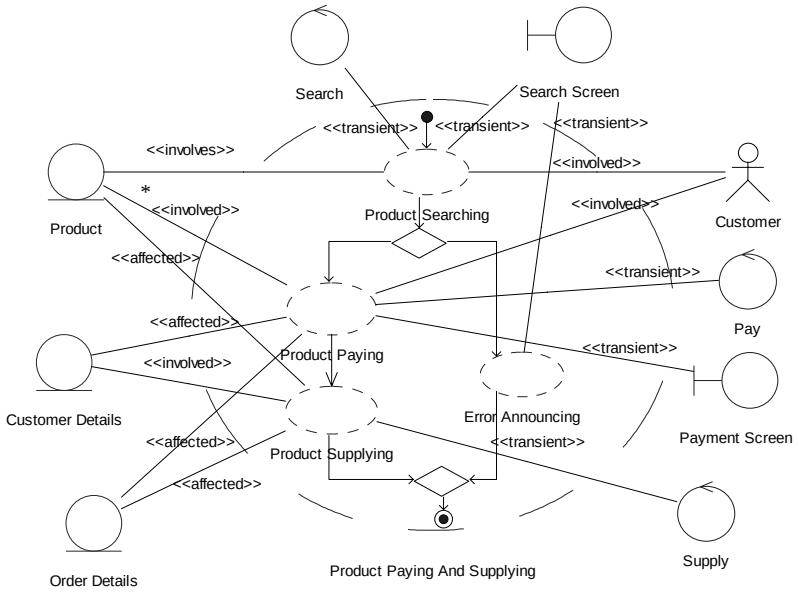
and Supplying, each of which is modeled in Figure 1 as a collaboration. From encapsulation and reuse perspectives, **Product Paying And Supplying** includes **Product Searching**, which realizes **Product Finding** as well as **Product Ordering**. All the three collaborations realize the **Product Ordering** use case. The transition between the abstract level of the system, i.e., the use cases, and the more refined description of the same system (the collaborations) is explicit through realization relations. Moreover, this transition is relatively simple since both use cases and collaborations refer to functional elements.

Each one of the collaborations that appears in Figure 1 can be in-zoomed to express its internal structure and behavior. The collaborations are classified into simple and compound collaborations. Compound collaborations are composed of other simple or compound collaborations and, hence, are modeled by composite structure diagrams. A *composite structure diagram* follows activity diagram notations for ordering and activating collaborations. A simple collaboration includes a single (possibly generic) scenario. Simple collaborations are modeled using interaction diagrams, i.e., sequence or collaboration diagrams. A simple or compound collaboration defines which objects can participate in the collaboration, how many objects of the same class can participate in the collaboration, and what their roles are. Using UML stereotypes, there are five possible roles:

1. The <<involved>> stereotype connects a collaboration to a class (or an actor) the objects of which can participate in the collaboration as unchangeable inputs.
2. The <<affected>> stereotype connects a collaboration to a class the objects of which can be affected during the collaboration. These objects exist before and after the collaboration occurrence, but their data or states are changed.
3. The <<created>> stereotype connects a collaboration to a class the objects of which are created during the collaboration.
4. The <<deleted>> stereotype connects a collaboration to a class the objects of which are destroyed during the collaboration.
5. The <<transient>> stereotype connects a collaboration to a class the objects of which are local to the collaboration.

Product Paying and Supplying from Figure 1, for example, is a compound collaboration which is composed of four simple collaborations, **Product Searching**, **Product Paying**, **Product Supplying**, and **Error Announcing**. First, **Product Searching** is executed, determining if the **Product** was found or not. If the searched **Product** was found, **Product Paying** is executed followed by **Product Supplying**. Otherwise, **Error Announcing** is activated. Either way, the end of **Product Supplying** or **Error Announcing** determines the end of the whole **Product Paying And Supplying** collaboration. Following the branching and merging notations of UML activity diagrams, Figure 2 describes the above in a composite structure diagram. In addition, Figure 2 specifies the object classes needed for the different collaborations. **Product Paying**, for example, uses a boundary object of type **Payment Screen** and a control object of type **Pay** as transient elements. One **Customer** is involved in any **Product Paying** process. During its execution, **Product Paying** also affects an entity object of type **Order Details** and an entity object of type **Customer Details**. Examples for these effects can be changing the order status in **Order Details** and adding the customer's credit card details in **Customer Details**.

Product Paying can also use any information from the two entity objects, for example the ordered amount from **Order Details** which is needed for calculating the final order price. **Product**, on the other hand, is only involved in **Product Paying**, enabling the access to the product price but disabling its change.



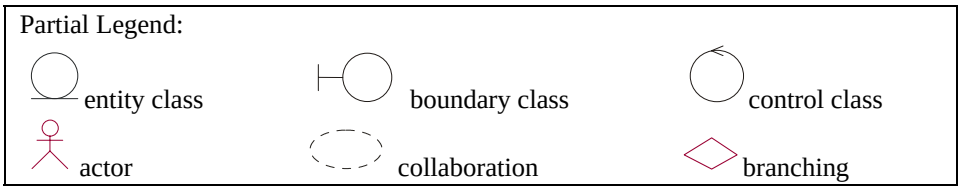


Figure 2. The composite structure diagram of **Product Paying and Supplying**

Following [11, 15] results, **Product Searching**, which is a simple collaboration, is expressed by the sequence diagram[2] in Figure 3. This diagram preserves the "interface" level described by the composite structure diagram in Figure 2. In other words, in this scenario **Search Screen** and **Search** are transient, while **Customer** and **Product** are only involved (used without being changed).

[2] For simplicity, the operation signatures in the sequence diagram are suppressed, not showing the operation parameters.

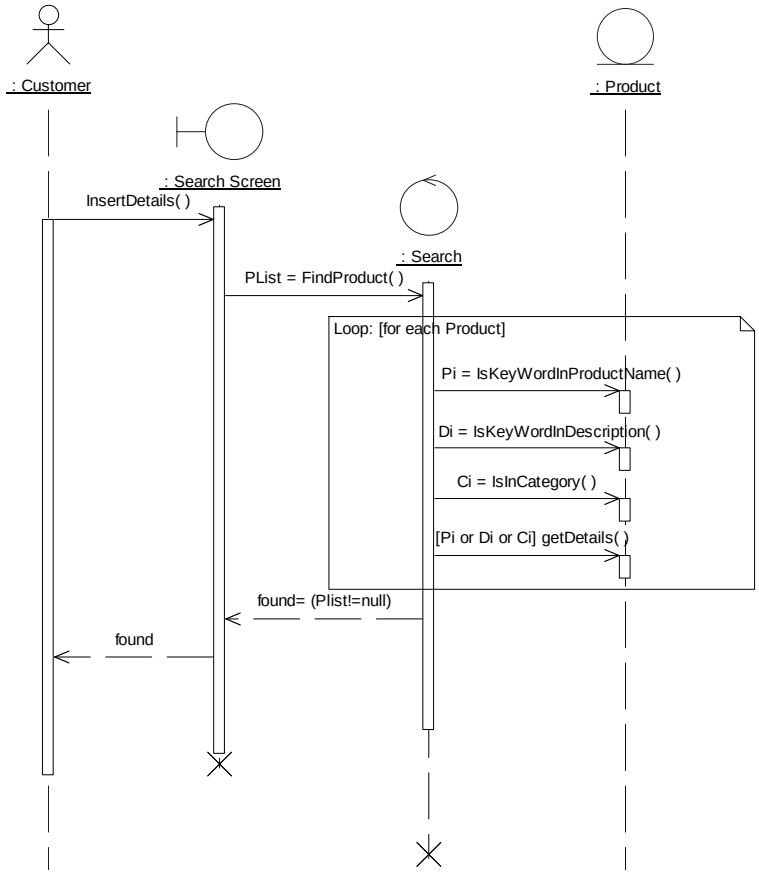


Figure 3. A sequence diagram describing **Product Searching**

The collaborations, which are described in composite structure diagrams and in interaction diagrams, induce a basic class diagram with its classes (boundary, control, and entity classes), associations, and operations. In the TLOOF approach, the structure of the system is developed to serve the functionality and not the other way, bridging the gap between the requirement analysis phase, which is behavior-oriented, and the design phase, which is architectural-oriented. Figure 4 is the class diagram induced from the three collaborations of the ordering system applying the following construction rules:

1. All the classes whose objects appear in any interaction diagram appear also in the class diagram.
2. An association between two classes in the class diagram exists if there is a message between two objects of these classes in an interaction diagram.
3. All the messages of an interaction diagram are interpreted into operations in the class diagram.

These construction rules follow the consistency rules required from class and interaction diagrams in a regular UML model [9]. For clarity purposes, the features

(attributes and operations) in Figure 4 are suppressed. After (automatically) creating the basic class diagram, the developers can improve it by adding attributes, associations, operations, etc.

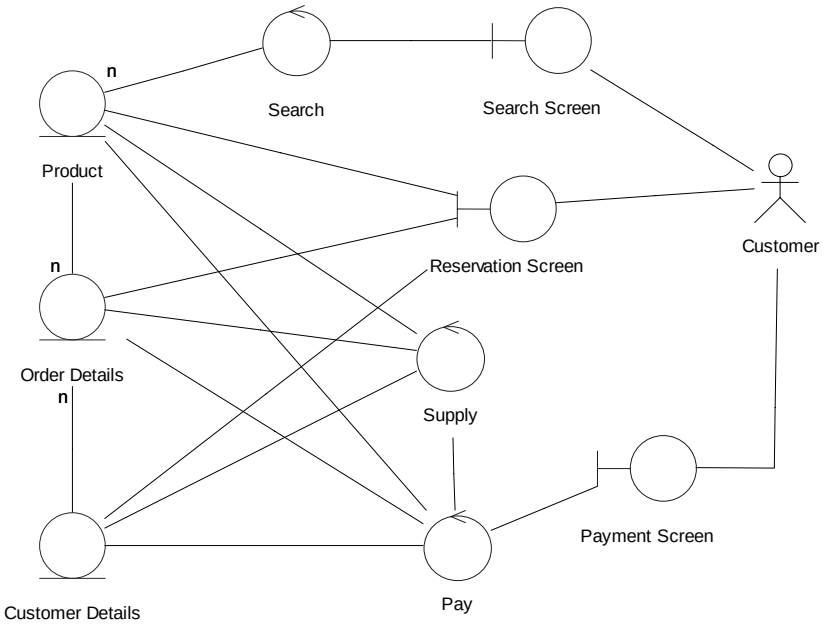


Figure 4. The induced class diagram for the ordering system

4 Consistency Rules for the TLOOF Approach

As noted, the TLOOF approach does not increase the vocabulary of UML, while better connecting the different views of the same system. Moreover, the explicit bindings of UML use case, class, and interaction diagrams in the TLOOF approach helps defining consistency rules between these views. This section defines and exemplifies the new consistency rules introduced by the TLOOF approach.

4.1 Consistency Rules within the TLOOF Diagram

As noted, the TLOOF diagram extends the use case diagram and, hence, all the use case diagram rules should be enforced in the TLOOF diagram as well. In particular, each use case in a TLOOF diagram should be connected either directly or indirectly to an actor. Indirect connections exist through inheritance relations, while direct

bindings use associations. In addition, the TLOOF approach defines the required realization relations between use cases and collaborations.

The TLOOF realization rule:
Each use case in a TLOOF diagram is realized by at least one collaboration.
Each collaboration in a TLOOF diagram realizes at least one use case.

Relating to vertical consistency, the TLOOF realization rule enables traceability of the system functional requirements. Each functional requirement, which is expressed by a use case, is realized by at least one collaboration, ensuring that no requirements are lost. Furthermore, the TLOOF realization rule also ensures that the implemented system, expressed by the collaborations, will not include additional, not requested functionality. In Figure 1, for example, the **Product Finding** use case (requirement) is realized by one collaboration, **Product Searching**, while **Product Ordering** is realized by three collaborations. Each collaboration realizes a requirement, except of **Product Searching** that realizes two requirements.

4.2 Consistency Rules of Composite Structure Diagrams

A composite structure diagram defines the interface of a collaboration: what are the classes whose objects participate in the collaboration and what are their multiplicities and roles. The relation between a collaboration and a class can be stereotyped by one of the following: involved, affected, created, deleted, or transient, each of which refers to a different possible effect of the collaboration on the class objects.

The compound collaboration refinement rule:
The declaration (interface) of a compound collaboration includes the declarations of its constituent collaborations. The inclusion order of the collaboration association stereotypes is affected (the most general), created, deleted, involved, and transient (the most particular).

The compound collaboration refinement rule regards to consistency between a composite structure diagram that describes a compound collaboration and the composite structure diagrams that describe its constituents. Hence, for example, if we were zooming-out from **Product Paying And Supplying**, shown in Figure 2, this collaboration would be connected to **Customer** via an involved-stereotyped association, to **Pay**, **Search**, **Supply**, **Payment Screen**, and **Search Screen** via transient-stereotyped associations, and to **Product**, **Order Details**, and **Customer Details** via affected-stereotyped associations. Furthermore, the multiplicity of **Product** in this collaboration would be many, while the multiplicity of all the other classes would be 1.

The actor participation rule:
An actor appears in a composite structure diagram if the actor is connected to a use case which is realized by that collaboration. This connection can be either directly via an association or indirectly by an inheritance relation.

The actor participation rule is derived from the vertical consistency requirement: if an actor is required for a use case, then it will be required for the collaborations that realize (implement) this use case. No contradictions should occur when refining the requirement specification expressed in a use case diagram to a more detailed design specification expressed in composite structure diagrams. In Figure 2, **Customer** is involved in **Product Searching**, since in Figure 1 there is an association between **Customer** and **Product Finding**, whose realization is **Product Searching**. Similarly, **Customer** is involved in **Product Paying** due to the fact that **Product Paying** is part of **Product Paying And Supplying** and the latter realizes **Product Ordering**, which is connected to **Customer** in the TLOOF diagram shown in Figure 1. If **Customer** were not connected directly to **Product Finding** in Figure 1 but through another use case, say **Product Handling**, from which **Product Finding** inherits, the **Customer** would be still involved in the composite structure diagram of **Product Searching**.

4.3 Consistency Rules between Composite Structure Diagrams and Interaction Diagrams

Four rules define the consistency required between composite structure diagrams and interaction diagrams. Three of these rules correspond to three of the five stereotypes of composite structure diagrams: created, deleted, and transient[3]. The fourth rule concerns that there will be no additional, redundant objects in the interaction diagrams, i.e., objects whose classes are not declared in the corresponding composite structure diagram.

The created object rule:
Objects of a class which is connected to a collaboration via a created-stereotyped association should be created (without deleting) in at least one related interaction diagram. Furthermore, the number of the created objects in a single interaction diagram should not exceed the corresponding class multiplicity in the collaboration.

[3] The two other stereotypes, affected and involved, require naming convention rules and, hence, are not defined as consistency rules.

The deleted object rule:
Objects of a class which is connected to a collaboration via a deleted-stereotyped association should be deleted (without creating) in at least one related interaction diagram. Furthermore, the number of the deleted objects in a single interaction diagram should not exceed the corresponding class multiplicity in the collaboration.

The transient object rule:
Objects of a class which is connected to a collaboration via a transient-stereotyped association should be created and deleted in at least one related interaction diagram. Furthermore, the number of the transient objects in a single interaction diagram should not exceed the corresponding class multiplicity in the collaboration.

The **Search Screen** and **Search** classes are connected via transient-stereotyped associations to the **Product Searching** collaboration in the composite structure diagram shown in Figure 2. In the sequence diagram which describes this collaboration, shown in Figure 3, one **Search Screen** object and one **Search** object are transient, i.e., created and deleted within the specific scenario. In other words, the effect of the sequence diagram on these objects corresponds (does not violate) the interface declared by the composite structure diagram.

The redundant object rule:
The class of each object that appears in an interaction diagram should appear also in the composite structure diagram of the corresponding collaboration. The class multiplicity in that collaboration is the maximum number of objects that appear in a single interaction diagram of that collaboration.

The redundant object rule ensures that there will be no objects that participate in an interaction diagram, while their classes are not declared in the collaboration interface. Figures 2 and 3 exemplify this rule: the class of each object that appears in Figure 3 appears also in Figure 2.

5 Summary and Future Work

The Top-Level Object-Oriented Framework (TLOOF) approach glues UML views by introducing the TLOOF diagram, which is actually an extension of the use case diagram with realized collaborations. The TLOOF diagram is refined into composite structure diagrams, each of which represents a separate collaboration. A composite structure diagram is refined by other composite structure diagrams in case of compound collaborations or by interaction diagrams in case of simple collaborations. This set of diagrams induces a connected graph with a single root, the TLOOF diagram, whose leaves are interaction diagrams. The connected graph enables smooth transitions from one aspect of the system to another without loosing the legibility and comprehension of the entire system. Seven rules, which can be easily implemented and checked, ensure that the UML models obtained in the TLOOF approach are consistent. This set of rules is complete, since it defines a consistency rule for each

element that appears in more than one diagram type. Contrarily to the translation and verification approaches for solving UML consistency and integration problems, the TLOOF approach enforces developing only consistent and integral UML models.

The TLOOF approach makes use of existing notions of UML, such as collaborations, realization relations, and composite structure diagrams. The associations between collaborations and classes are classified using stereotypes, a UML build-in extension mechanism. While not extending the UML vocabulary, the TLOOF approach provides the missing glue for UML views and enables checking model consistency and integrity. Furthermore, the TLOOF approach bridges the gap between the requirement analysis and design stages, enabling requirement traceability. Indeed, checking the comprehension of regular UML models vs. TLOOF models (i.e., UML models that apply the TLOOF approach) on a small group of undergraduate information system students, the TLOOF models were found to be more comprehensive in their description of system behavior and more supportive in requirement traceability.

Further research is planned to deal with overlapping interactions, synchronization points of collaborations, and distribution of collaborations. A series of experiments is also planned to verify the comprehension and easiness of developing UML models in the TLOOF approach.

References

1. Baresi, L., Pezze, M.: On Formalizing UML with High-Level Petri Nets. Concurrent Object-Oriented Programming and Petri Nets (2001) 276-304.
2. Bowman, H., Steen, M., Boiten, E.A., Derrick, J.: A Formal Framework for Viewpoint Consistency. Formal Methods in System Design 21 (2) (2002) 111-166.
3. Bodeveix, J.P., Millan, T., Percebois, C., Le Camus, C., Bazex, P., Feraud, L., Sobek, R.: Extending OCL for Verifying UML Models Consistency. Workshop on Consistency Problems in UML-based Software Development, 5th International Conference on the Unified Modeling Language- the Language and its applications (UML'2002), Dresden, Germany (2002) 75-90.
4. Chiorean, D., Pasca, M., Carcu, A., Botiza, C., Moldovan, S.: Ensuring UML models consistency using the OCL Environment. Workshop on OCL 2.0 - Industry standard or scientific playground?, 6th International Conference on the Unified Modeling Language - the Language and its applications (UML'2003), San Francisco (2003), http://i11www.ira.uka.de/~baar/oclworkshopUml03/papers/06_ensuring_uml_model_consistency.pdf
5. Dori, D.: Object-Process Methodology - A Holistic Systems Paradigm. Springer Verlag, Heidelberg, NY (2002).
6. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Testing the Consistency of Dynamic UML Diagrams. Proc. 6th International Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena CA (2002), http://www.uni-paderborn.de/cs/ag-engels/Papers/2002/EngelsHHS-IDPT02.pdf
7. Engels, G., Kuster, J. M., Groenewegen, L., Heckel, R.: A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In V. Gruhn (ed.): Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9). ACM Press, Vienna Austria (2001) 186-195.

8. Fowler, M., Scott, K.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd edition, Addison-Wesley (2003).
9. Gomaa, H., Wijesekera, D.: Consistency in Multiple-View UML Models: A Case Study. Workshop on Consistency Problems in UML-based Software Development II, 6th International Conference on the Unified Modeling Language- the Language and its applications (UML'2003), San Francisco (2003) 1-8.
10. Große-Rhode, M.: Integrating Semantics for Object-Oriented System Models. 28th International Colloquium on Automata, Languages and Programming (ICALP 2001), Crete, Greece, Lecture Notes in Computer Science 2076 (2001) 40-60.
11. Hahn, J., Kim, J.: Why Are Some Diagrams Easier to Work With? Effects of Diagrammatic Representation on the Cognitive Integration Process of Systems Analysis and Design. ACM Transactions on Computer-Human Interaction, 6 (3) (1999) 181-213.
12. Mens, T., Van Der Straeten, R., Simmonds, J.: Maintaining Consistency between UML Models Using Description Logic. Workshop on Consistency Problems in UML-based Software Development II, 6th International Conference on the Unified Modeling Language- the Language and its applications (UML'2003), San Francisco (2003) 71-77.
13. Nentwich, C., Emmerich, W., Finkelstein, A., Ellmer, E.: Flexible consistency checking. ACM Transactions on Software Engineering and Methodologies 12 (1) (2003) 28-63.
14. Object Management Group. Unified Modeling Language Specification – version 1.4. ftp://ftp.omg.org/pub/docs/formal/01-09-67.pdf
15. Otero, M.C., Dolado, J.J.: An Initial Experimental Assessment of the Dynamic Modeling in UML. Empirical Software Engineering 7 (2002) 27-47.
16. Peleg, M., Dori, D.: The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods. IEEE Transaction on Software Engineering 26 (8) (2000) 742-759.
17. Rasch, H., Wehrheim , H.: Consistency Between UML Classes and Associated State Machines. Workshop on Consistency Problems in UML-based Software Development, 5th International Conference on the Unified Modeling Language- the Language and its applications (UML'2002), Dresden, Germany (2002) 46-60.
18. Reinhartz-Berger, I., Dori, D.: OPM vs. UML – Experimenting Comprehension and Construction of Web Application Models. Accepted Empirical Software Engineering (EMSE).
19. Tilley, S., Huang, S.: A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. Proceedings of the 21st annual international conference on Documentation, San Francisco, CA (2003) 184-191.
20. Tun, T., Bielkowicz, P.: A Critical Assessment of UML using an Evaluation Framework. 8th CAiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'03) (2003) 29-37.
21. Uchitel, S., Kramer, J. and Magee, J. Synthesis of Behavioral Models from Scenarios. IEEE Transactions on Software Engineering 29 (2) (2003) 99-115.