

Consistent Interaction of Software Components

Gregor Engels and Jochen M. Küster
Department of Computer Science
University of Paderborn
D-33095 Paderborn, Germany
engels@upb.de, jkuester@upb.de

and
Luuk Groenewegen
Leiden Institute of Advanced Computer Studies
Leiden University
P.O. Box 9512, NL-2300 RA Leiden, The Netherlands
luuk@liacs.nl

ABSTRACT: Constructing complex software systems by integrating different software components is a promising and challenging approach. With the functionality of software components given by models it is possible to ensure consistency of such models before implementation in order to successfully build the system. Models consisting of different submodels, the absence of an overall formal semantics and the numerous possibilities of employing models requires the development of techniques ensuring the consistency. In this paper, we discuss the issue of consistency of models made up of different submodels proposing a concept for the management of consistency. Consistency management relies on a concept of consistency and a process for ensuring consistency of models. We introduce a consistency concept for software components modeled in the Unified Modeling Language (UML) and devise suitable consistency checks. On this basis, we propose a process how to locate and resolve inconsistencies, thus ensuring the consistency of models and by that the consistency of component-based systems derived from those models.

I. INTRODUCTION

Software systems usually are large and complex. Because of the size as well as because of the complexity, engineering such a software system is not a task to be taken lightly. Therefore many variants of the well known and generally successful divide-and-conquer approach are in use in the software engineering process.

One such an approach consists of building the software system from smaller software systems, sometimes referred to as subsystems or more usually as components. A component is a software system or program on its own, of such a size and complexity that engineering it is a reasonable effort, in general far less difficult than engineering the whole system. Such a component itself may be composed of other, smaller and simpler components.

In order to deal with the complexity of software systems, the use of models is nowadays a well-established approach.

Models are used within the software engineering process for means of communication, documentation and requirements capture and allow the abstraction from implementation details such as programming language and computer platform. With the functionality of component descriptions given in form of models, the foundation for the ability of reasoning about the nature of complex systems made up of simpler components is in place.

Successfully integrating components on the model level gives rise to the following problem of model consistency: The overall model made up of a number of component models has to be consistent and not contradictory. Otherwise, an implementation of the overall model will not be feasible, thereby making the overall model useless.

In this paper, we discuss the issue of consistency of models made up of different submodels proposing a concept for the management of consistency. Consistency management relies on a concept of consistency and a process for ensuring consistency of models. We introduce a consistency concept for software components modeled in an extension of UML [18] and devise suitable consistency checks. The paper is organized as follows: First, we discuss the issue of modeling of components. Then, in section III, we discuss consistency of models and, in section IV, consistency management in general. In section V, we propose a general methodology for ensuring consistency of models which we apply to the problem of ensuring consistency of component models in section VI.

II. MODELING OF COMPONENTS

Software systems or programs comprise many different, partially overlapping aspects - such as the static or structural ones: data and architecture, and the dynamic or process-like ones: visible behavior or interface behavior, hidden behavior or functionality, and interaction, the aspect covered by communication, collaboration, cooperation and coordination. Also components comprise all these aspects, as they are software systems, too. Composing the

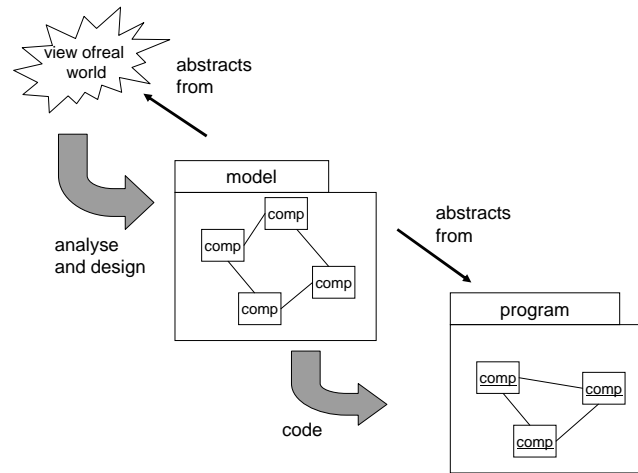


Fig. 1. A model in relation to reality and to a program composed of components

components into a larger software system then consists of putting them together in a sufficiently careful manner, such that with respect to all aspects involved the components fit together. In order to be able to compose the components as required, one wants to reason about and to analyse possible compositions of the components, one wants to understand their eventual configuration, one wants to be convinced of their consistency. In this paper, we are particularly interested in the consistency of interactions performed by the components.

It is through modeling software, that software can be studied best in order to enable reasoning about it, analysing it, understanding it. As, in the case of software components, we want to study their composition with respect to all aspects involved, a model of such software should reflect the components as well as their composition, thereby covering all relevant aspects in a sufficiently precise manner. Figure 1 visualizes the modeling situation for software consisting of components.

The above representation clarifies that the details of the interaction of the program components can be studied through the details of the interaction of the model components, provided that two conditions are fulfilled. The first condition is, that the meaning of models is fixed in such a way that reasoning, analysis and understanding of models is possible. The second condition is, that the abstraction from program to model indeed preserves all aspect details relevant to the interaction of program components. This means that the interaction of program components generated from model components is in principle the same interaction as the one we study on the model level.

Regarding the first condition, the meaning of models is usually fixed by the use of a modeling language. Such a modeling language allows us to construct models such that the meaning of models can be communicated. As the modeling language determines the semantics of models it is crucial for the ability of constructing useful models. The

modeling language insofar provides the means of abstraction from the real world and the program, concentrating on certain aspects and not considering other aspects. Typically, a model consists of several submodels, each submodel concentrating on specific aspects. Concerning the second condition, we observe that the nature of the modeling language determines the kind of abstraction and thereby also the preservation of aspect details.

As a consequence, the choice and the characteristics of the modeling language are crucial to the success of modeling. How can we determine whether a modeling language is suitable or not?

Given a problem domain and an understanding of the system to be built, the modeler has a certain *generic conceptual model* in mind. This generic conceptual model determines the aspects to be modeled and thus the kind of abstraction. The chosen modeling language must then support the same aspects.

In Figure 2, the relationship of the generic conceptual model and the modeling language is illustrated. Abstraction from the real world and the program yields a generic conceptual model. The generic conceptual model concentrates on certain aspects of the real world and the program. The chosen modeling language must support the construction of concrete models and consequently must support the same aspects as the generic conceptual model. As a consequence, the set of aspects the generic conceptual model includes serves as a classification criteria for the choice of the semantics of the modeling language. In particular, each individual aspect serves as a requirement for the semantics of the modeling language: It must be possible to express in a model the aspect of the generic conceptual model. During analysis and design, a concrete model is constructed using the chosen modeling language.

The previous ideas can be exemplified for the problem of modeling component-based software choosing the Unified Modeling Language [18] as our modeling language. We

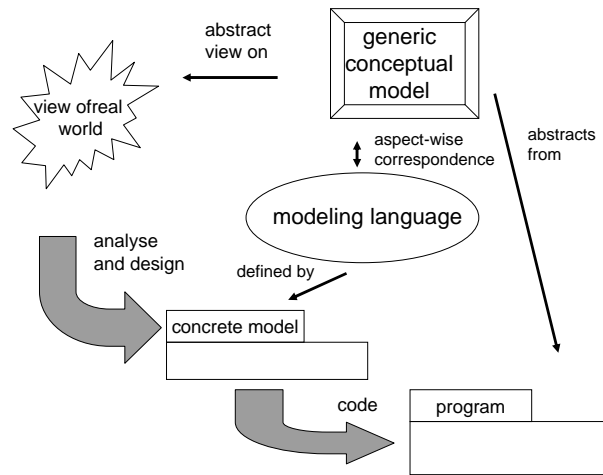


Fig. 2. Conceptual model and modeling language

first note that, as a general-purpose modeling language, the UML covers all the above mentioned aspects and should therefore be in principle suitable for studying the interaction of program components at the model level. The UML offers a number of different sublanguages and a typical UML model consists of several submodels, each submodel concentrating on important aspects of the system to be modeled.

Concerning the ability of analysing the interaction of component-based software at the model level, a fixed meaning of models is crucial. Deplorably, the UML does not have a formally defined semantics and therefore the reasoning and analysis of component models are not straightforward. In order to apply UML to the successful modeling of components, the deficiencies of UML with respect to consistency of different submodels have to be coped with: Given a component model in UML consisting of submodels each concentrating on different aspects, it has to be ensured that the overall model has a proper semantics and is not contradictory. In the following three sections, we therefore discuss the notion of consistency in general, proposing a general methodology for consistency management of UML models. This methodology is then applied in order to ensure consistent interaction of software components.

III. CONSISTENCY

In this section, we introduce and discuss the notion of consistency. First, we explain general concepts of consistency within software engineering and introduce the terms of a consistency problem and consistency condition. Then we explain characteristics of consistency problems.

A. General concepts

The use of models consisting of different submodels within software development has numerous advantages. Different persons may work on different submodels simultaneously driving forward the development of the system.

Different types of submodels allow the separation of different aspects of the system to be built such as structural aspects or dynamic behavior of system components.

However, making use of different submodels also involves drawbacks. In a traditional approach, there exists only one model of the system. This model can then be transformed during coding into a running software product. In the case of a collection of submodels, this is not as easy anymore because one needs to describe which submodel is transformed into which part of the code. This gives rise to the problem of different parts of the code not working together as one wishes leading to a system not functioning. In order not to run into such problems, it has to be ensured that different submodels are compatible with each other or consistent on the model level.

Different submodels of a model are called consistent if they can be integrated into a single model with a proper semantics. As a consequence, consistency of submodels ensures the existence of an implementation: if consistency is ensured, an implementation of submodels is obtained by implementing the integrated model. Otherwise, such an integrated model and implementation might not exist.

The previous definition allows us to discuss consistency on a high level of abstraction. For particular types of models consisting of specific submodels with a specific semantics, this definition must be refined by resolving what it means to integrate submodels into a single model. Submodels that can be integrated are then called consistent.

Depending on the models used within software development, we can distinguish between different scenarios of consistency. If the semantics of each submodel is defined in terms of a common system model [3], then consistency of submodels can be refined to be the existence of a non-empty system model. Otherwise, if the semantics of each submodel is defined in its own semantic domain, then these semantic domains must be integrated into a common semantic domain. Consistency can then be refined to be the

possibility of integrating the semantics of submodels.

Whenever submodels of a model are not consistent, we speak of a *consistency problem*. If models consisting of different submodels are applied within software development, there have to be techniques in order to ensure the consistency of the overall model and to resolve inconsistencies within the model. In order to resolve potential inconsistencies, one has first to get a clear idea of when a model is consistent or not. For this purpose, one uses *consistency conditions*: A consistency condition defines whether a model is consistent or not. Given a model consisting of different submodels and a number of consistency conditions, it can now be determined whether the model is consistent or not. Consistency conditions insofar provide the basis of defining consistency of models. Consistency conditions depend heavily on the modeling language used and on the development process because they both determine the aspects described.

B. Characteristics of Consistency Problems

Consistency problems can be characterized on the one hand according to the situation they occur and on the other hand depending on the consistency condition.

One problem of consistency arises in cases where a specification consists of different parts because a system is modeled from different viewpoints [1], [8]. This allows the concentration on different aspects in different models. However, different viewpoint specifications must be consistent and not contradictory, because the implementation of such an inconsistent specification would otherwise be unfeasible. This type of consistency problem we will call *horizontal consistency*.

One important prerequisite for this type of consistency problem to arise is that there is a certain viewpoint overlap, also called *correspondence* [1]. This means that viewpoint specifications are not completely independent from each other but also describe common aspects.

Another quite different problem of consistency arises when a specification is transformed into another refined specification. It is then desirable that the refined specification is consistent with the previous specification, in order to keep the overall specification consistent. This type of consistency problem we will call *vertical consistency*. Vertical consistency problems are induced by a development process which prescribes how and when models are iteratively refined.

A quite different characterization is obtained by looking at the consistency condition for a consistency problem. *Syntactic consistency* ensures that a model is consistent with respect to the syntax and is ensured by formulating consistency conditions on the syntax of models. Concerning horizontal consistency problems, syntactic consistency ensures that the overall model consisting of submodels is syntactically correct. With regards to vertical consistency problems, syntactic consistency ensures that changing of one

part of the model within the development process still results into a syntactically correct model.

Semantic consistency ensures that a model is consistent with respect to its semantics and is ensured by formulating consistency conditions on the semantics of models. With respect to a horizontal consistency problem, semantic consistency requires models of different viewpoints to be semantically compatible with regards to the aspects of the system which are described in the submodels. For vertical consistency problems, semantic consistency requires that a refined model is semantically consistent with the model it refines.

IV. CONSISTENCY MANAGEMENT

Ensuring the consistency of models within a software development process gives rise to the need of consistency management. In this section, we introduce and discuss the notion of consistency management identifying important characteristics. Along these characteristics we categorize related work relevant to ensuring consistency of models.

A. The notion of consistency management

Given a set of models and a development process, it arises the question how to ensure consistency of such models within the development process. Obviously, this requires an understanding or concept of consistency and the definition of consistency conditions, the specification when consistency conditions are checked and what to do in the case of inconsistencies in order to resolve these. We thus need a sort of management of consistency and introduce the term *consistency management*: *Consistency management* is the technique which has as its goal to ensure the consistency of models within a software engineering process.

In order to generally ensure the consistency of models, the foundation of consistency management is the ability to decide whether two models are consistent or not. As a consequence, consistency management relies on consistency conditions. However, before being able to define consistency conditions, it must be clarified which models are inconsistent because they cannot be semantically integrated. The basis for consistency management is therefore a careful inspection of models used and their semantics, leading to a definition of consistency.

Given such a definition of consistency, consistency management must enable the definition of consistency conditions for models. This includes the identification of potential consistency problems and the decision when models should be consistent with respect to which consistency condition.

Once these foundations for consistency management have been fixed and consistency conditions defined, consistency management also involves the embedding of dealing with consistency within a development process. For this purpose, given a set of consistency conditions and a set of models, consistency management should describe for each

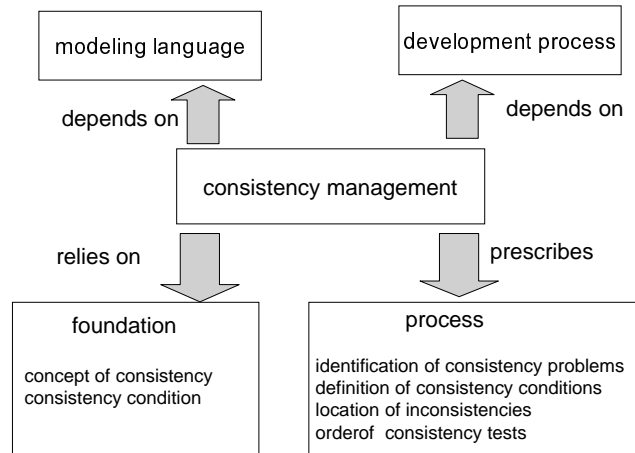


Fig. 3. Consistency Management

consistency condition how a consistency check can be performed on models in order to ensure consistency of models with respect to the consistency conditions.

In Figure 3 consistency management is illustrated. Consistency management depends heavily on the modeling language used mirroring the underlying generic conceptual model. The modeling language always includes a certain form of syntactic consistency ensured by the syntactical correctness of models. It furthermore prescribes a certain degree of semantic consistency ensured by semantics of models. Depending on the underlying generic conceptual model and the development process, consistency management must ensure additional syntactic and semantic consistency. In order to achieve this, consistency management relies on a foundation for consistency consisting of a concept of consistency including consistency conditions. It prescribes a process including the identification of consistency problems and definition of consistency conditions. On the basis of these, consistency management involves the location of potential inconsistencies and the performing of consistency checks during the overall software development process.

B. Related approaches

Existing approaches can be categorized into several categories. The first category contains approaches where a particular consistency problem is tackled. For instance, Fradet et al. [10] propose an approach to consistency checking of diagrams consisting of nodes and edges with multiplicities. They distinguish between generic and instance graphs and define the semantics of a generic graph to be the set of instance graphs that fulfill the constraints of the generic graph. Consistency is then defined to be equivalent to the semantics of the generic graph being not an empty set. Consistency checking is then performed by solving a system of linear inequalities derived from the generic graph. Also in this category falls the approach by Li et al. [15] who ana-

lyze timing constraints of sequence diagrams for their consistency solving systems of linear inequalities.

Another category can be seen in approaches that achieve consistency of object-oriented models by completely formalizing them, thereby integrating all models into one semantic domain. Moreira and Clark [16] translate object-oriented analysis models to LOTOS in order to detect inconsistencies. Cheng et al. [4] formalize OMT models in terms of LOTOS specifications. Using these specifications, they perform consistency analysis using tools for state exploration and concurrency analysis. Grosse-Rhode [12] integrates all models by translating them into transformation systems. The problem involved with completely formalizing models is that the application is then restricted to a certain problem domain mirrored by the choice of semantic domain.

A third category can be seen in approaches that deal with consistency of models that are not object-oriented. Zave et al. [21] define consistency based on a translation into logics and define a set of partial specifications to be consistent if and only if its composition is satisfiable. Their approach therefore requires that models are given a semantics in form of logics. Boiten et al. [2] define consistency on the basis of development relations and define a set of partial specifications to be consistent if and only if a common development relation exists. This approach requires the existence of a formal semantics for models and the concept of development relations defined for models used within the development process.

Another, quite different, category of related work can be seen in approaches that deal with inconsistency management [17] [11]. Rather than trying to achieve complete consistency, these approaches tackle the problem of managing inconsistencies. This management is based on the location of inconsistencies and the appropriate handling (resolving of inconsistency or tolerating it). Concentrating on the process of consistency management, they assume that the foun-

dation of consistency management in terms of consistency conditions is already in place.

In the following section, we present an approach to consistency management based on the idea of partially formalizing the model for enabling consistency checks. As a consequence, our approach can be regarded as a combination of the approaches in the first and the second category. As our approach also comprises the idea of consistency management within a development process, it is also related to the fourth category although the idea of tolerating inconsistencies is not in focus.

V. A METHODOLOGY FOR CONSISTENCY MANAGEMENT OF UML MODELS

Having discussed the issue of consistency and consistency management, we now turn our attention to the modeling of component-based systems. We will model these systems using UML because UML is nowadays a well-accepted modeling language. Despite its numerous advantages such as user-friendliness by visual models, there are currently also certain drawbacks when it comes to establishing consistency of models. We can describe the situation as follows:

Due to the unclear semantics and different employments of UML within the development process, a general concept of consistency management for UML models is missing [5]. Concerning syntactic consistency, this type of consistency is partly achieved by defining well-formedness rules on the metamodel. Due to the absence of a generally-accepted formal semantics, semantic consistency is currently not well-supported. Nevertheless, semantic consistency is of great importance and must be dealt with. Another problem is the informal development process followed which leads to the need of flexible notions of consistency. We thus conclude that consistency conditions for UML models depend on the problem domain the language is applied to and the development process followed.

In the following, we will propose a general methodology for consistency management of models exemplified for the case of UML models. This methodology will then be applied to the problem of consistency when modeling components, focusing on consistency of interaction of components.

Our approach to consistency management of UML models is based on the following observation: The fundamental question to answer when given a collection of UML models is whether there exist a common semantic interpretation of them. Although a formally defined semantics does not exist, it is still possible to restrict oneself to certain aspects of models and then determine whether an integration or common semantic interpretation exists.

The concept of our approach is illustrated in Figure 4. Submodels used within development overlap in a number of aspects. For ensuring consistency, submodels are integrated into a common semantic domain, that supports these

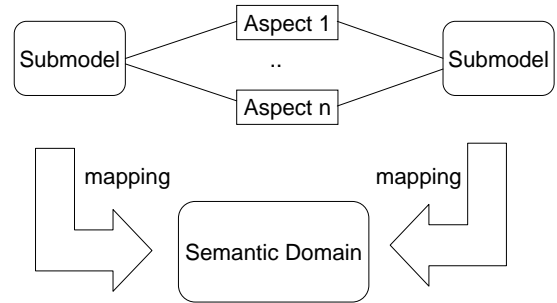


Fig. 4. Concept of our approach

overlapping aspects. If a concrete model cannot be integrated, then it is inconsistent.

We therefore propose the following methodology for establishing consistency:

Step 1: Identification of generic conceptual model, aspects, modeling language. The first step in the methodology is to establish a generic conceptual model of the problem domain and those systems one wishes to model. This can only be done on an informal basis and involves abstracting from minor aspects and concentrating on important aspects the model should support. The aspects to be captured by the model must be elaborated to a classification of aspects. The modeling language has to be fixed and the development process is defined as to which submodels are to be used in which step of the development process. Furthermore, consistency requirements for the application domain must be identified and stated in textual form.

Step 2: Identification of consistency problems. The basis for the identification of consistency problems is obtained by relating submodels to aspects and thereby discovering which submodels model the same aspects of the system. Due to this common description of aspects in different submodels (also called overlap of aspects) consistency problems may occur if the description of aspects contains contradictions. These consistency problems are identified and stated using a prototypical situation leading to the consistency violation. Concerning each prototypical situation, we relate it to the development process for deciding when such a prototypical situation may occur. Furthermore, we decide in how far the submodels must be syntactically consistent and we define syntactic consistency conditions.

Step 3: Choice of a semantic domain. For each consistency problem identified, we choose an appropriate semantic domain. This semantic domain must support those aspects that lead to the consistency problem (i. e. the aspects where submodels overlap). Furthermore, the semantic domain should provide tool support in order to facilitate consistency checks. In order to validate that the semantic domain is indeed appropriate we translate the prototypical situations into the semantic domain.

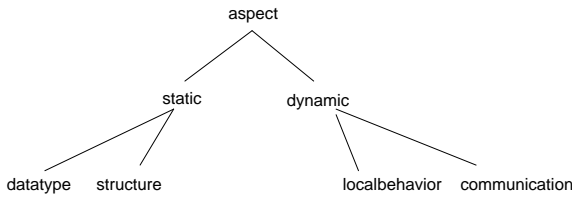


Fig. 5. Aspects of component-based systems

Step 4: Definition of partial mapping. All aspects of the model that lead to the identified consistency problem must be mapped into the semantic domain. The definition of the partial mapping is crucial for the correctness of later defined consistency conditions and no aspects of the model should be left out that influence the consistency of the model. On the other hand, only those aspects of the model should be mapped into the semantic domain that are important for the consistency because otherwise the later analysis may get too complex.

Step 5: Specification of consistency conditions. In the language of the semantic domain consistency conditions must be formulated that ensure the establishment of consistency within the model. Here, also the general consistency requirements for the conceptual model must be taken into account. The order of consistency test to be performed by grouping the consistency conditions according to their efficiency must be determined and fixed within the development process.

Step 6: Location of inconsistencies and analysis of potential inconsistencies. Given a concrete model consisting of submodels, the prototypical situations specified earlier are used to locate potential inconsistencies. This is done by finding all instances of a prototypical situation in the concrete model. Using the analysis techniques of the semantic domain the previously formulated consistency conditions must be analyzed for models mapped into (the language of) the semantic domain.

VI. APPLICATION TO COMPONENTS MODELED IN UML

In this section, we will apply the general methodology for consistency management in order to ensure the consistency of models for component-based systems.

A. Identification of generic conceptual model, aspects and modeling language

The basis of our methodology is firstly the identification of the generic conceptual model i. e. the identification of aspects, secondly the choice of the modeling language, thirdly the definition of the development process and fourthly the identification of consistency requirements.

Concerning the modeling of components, we recall our discussion of modeling software components and elaborate the aspects to a classification of aspects, see Figure 5. In

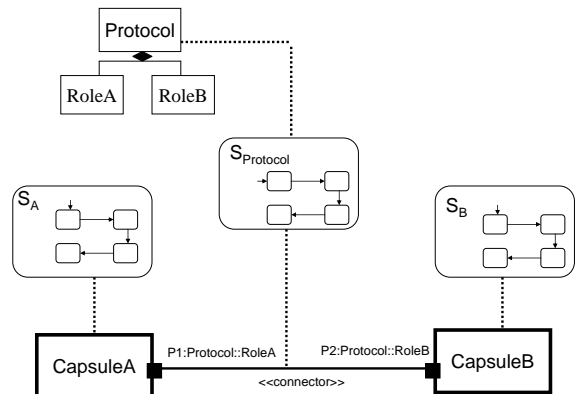


Fig. 6. Submodels and their usage in UML-RT

general, we can distinguish between static and dynamic aspects. The static aspects can be divided into a structure aspect describing the structure of the system and the data type aspect dealing with the definition of data types. With respect to the dynamic aspects, we can distinguish between communication and local behavior.

Concerning the modeling language, we already stated that we will use UML. More precisely, we concentrate on UML-RT [20], a profile of UML for modeling real-time systems and incorporating the concept of components called capsules. In the following, we shortly describe concepts of UML-RT and then we relate submodels used within UML-RT to the previously identified aspects, thereby showing that UML-RT indeed covers all the identified aspects.

UML-RT is an extension of the UML by introducing the notions of *capsules*, *ports*, *connectors*, *protocols* and *protocol roles*. Originally targeted at enabling modeling of complex real-time systems, UML-RT has also been seen as a candidate for modeling software architectures [19] and for modeling concurrent systems in general. In the following, concepts of UML-RT are explained in detail.

A *capsule* is a stereotyped active class and is used for modeling a self contained component of a system. Other than ordinary classes, capsule operations can only be called from within the capsule. For communication with other capsules a capsule may have one or more ports through which it is interconnected to other capsules via *connectors*. A connector is an association between capsules. It represents a hardware connection via which capsules communicate by sending and receiving signals. These signals enter or leave a capsule at a port. A *port* realizes a *protocol role* which specifies the signals sent and received via the port. One or more protocol roles form a *protocol*.

From the point of view of behavioral modeling, a statechart may be associated to a capsule. A capsule statechart describes how the capsule reacts to signals received via its ports and when signals are sent via its ports. State transitions of capsule statecharts may also include calls of cap-

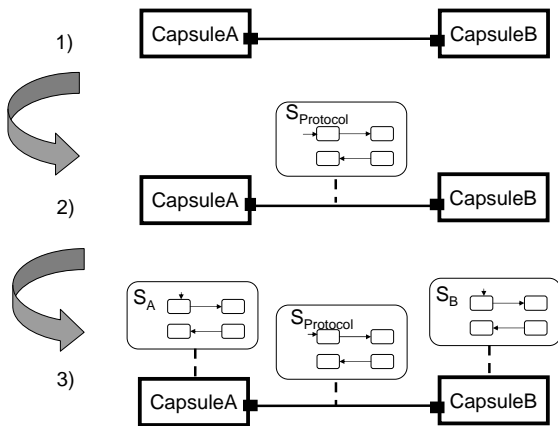


Fig. 7. A simplified development process

sule operations. For protocols, there exist also the possibility of modeling all valid sequences of message exchanges in a protocol statechart. The protocol statechart therefore expresses requirements rather than specifying the implementation of a protocol.

In Figure 6, concepts of UML-RT are illustrated using a very simple example consisting of two capsule instances **CapsuleA** and **CapsuleB** connected by a connector via the two ports **P1** and **P2**. The ports are bound to the protocol roles **RoleA** and **RoleB**, respectively. Each capsule is associated to a capsule statechart, S_A and S_B respectively, describing the dynamic behavior of the capsule. The protocol statechart $S_{Protocol}$ is associated to the connector between the two capsules.

Regarding the suitability of UML-RT for modeling component-based systems, we first relate submodels of UML-RT to the aspects. We note that data type and structure aspect is covered by the class diagram and the capsule collaboration diagram of UML-RT. The communication aspect is covered by the protocol statechart and the capsule statechart. The latter also covers the aspect of local behavior.

Concerning the use of submodels within the development process, we will assume a simple development process consisting of three steps: In the first step, a capsule collaboration diagram is used to model the structure of the system and for each capsule the data type aspect is modeled in a common class diagram. In the second step, protocol statecharts may be added to the model describing the communication aspect with respect to the allowed message exchanges via connectors. In the third step, capsule statecharts are defined for each capsule describing the communication and local behavior aspect of capsules. This simplified development process is illustrated in Figure 7, for space reasons the common class diagram is not shown.

With regards to consistency requirements, we identify the property of deadlock freeness and protocol conformance. Concerning deadlock freeness, the communication resulting from the interaction of capsules should be dead-

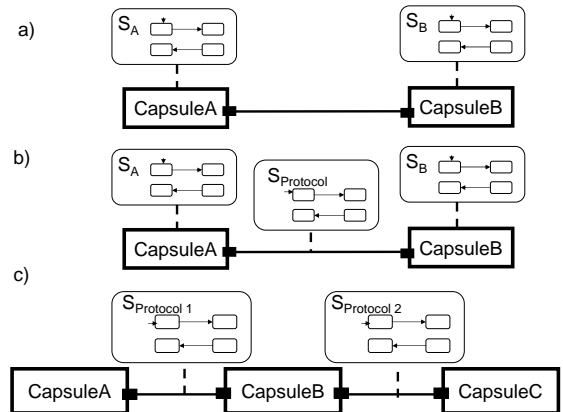


Fig. 8. Prototypical situation for consistency problems in UML-RT

lock free. Protocol conformance requires the conformity of capsule interaction with the protocol specified in the protocol statecharts associated to the connector between the capsules in focus.

B. Identification of consistency problems

The basis for the identification of consistency problems is a description of modeling aspects each submodel covers. As we are particularly interested in ensuring consistent interaction of components we restrict ourselves to the communication and interaction aspect. In the following, we identify consistency problems, describe prototypical situations, relate the prototypical situations to the development process and discuss the issue of syntactic consistency for the identified consistency problems.

In the case of UML-RT, we can group the submodels to the modeling aspects they concern finding out that capsule statechart and protocol statechart both relate to the communication modeling aspect.

After identification of the possible aspect overlap of capsule statecharts and protocol statecharts, we now look at this problem in detail. We realize that the following two consistency problems may occur:

- a) the consistency problem between two capsule statecharts
- b) the consistency problem between two capsule statecharts and a protocol statechart

In Figure 8, a prototypical situation for each of the two consistency problems a) and b) is presented. These prototypical situations are used later for locating potential consistency problems. In our case, the first consistency problem may occur if two capsule statecharts for two capsules are defined and these capsules are connected via a connector. The second consistency problem occurs, if additionally a protocol statechart is associated with the connector that connects the two capsules. In order to point out the importance of these prototypical situations, consider c) in Fig-

ure 8. Although the protocol statecharts both describe the communication aspect of the system, a consistency problem does not occur assuming no additional conformance requirement between the two protocols associated to different connectors.

The prototypical situations are related to the development process in the way that both consistency problems arise within the third step of the development process. Considering the development process, this is not surprising as only in the third step there are two submodels describing the communication and interaction aspect.

Both consistency problems have syntactic and semantic characteristics. Concerning syntactic consistency, syntactical consistency conditions in form of well-formedness rules for the syntax of models can be formulated in order to enable a common semantic interpretation. Possible rules are that the set of signals sent in S_A must be a subset of the set of signals received in S_B . If the submodels are syntactically consistent, semantic consistency must still be established because not every syntactically consistent model is necessarily semantically consistent. The following three steps of the methodology deal with semantic consistency.

C. Choice of a semantic domain

For each consistency problem, we have to choose an appropriate semantic domain which enables to determine whether two models may be integrated semantically. It is important that the semantic domain supports the modeling aspects in which the two models overlap. Concerning both consistency problems, the modeling aspect of overlap is communication. We choose as a semantic domain CSP [14] which provides a mathematical model for concurrency based on a simple programming notation and supported by tools [9]. In fact, the existence of *language* and *tool support* are most important to our aim of *specifying* and *verifying* consistency constraints, despite the existence of more expressive mathematical models. Next, we briefly review the syntax and semantics of the CSP processes we are using.

Given a set \mathcal{A} of actions and a set of process names \mathcal{N} , the syntax of CSP is given by

$$P ::= \text{STOP} \mid \text{SKIP} \mid a \rightarrow P \mid P[A \mid B]P \mid P \sqcap P \mid P \sqbox P \\ \mid P \setminus a \mid pn$$

where $a \in \mathcal{A}$, $A, B \subseteq \mathcal{A}$, and $pn \in \mathcal{N}$.

Process names are used for defining recursive processes using equations $pn = P$. The interpretation of the operations is as follows. The processes STOP and SKIP represent, respectively, deadlock and successful termination. The prefix processes $a \rightarrow P$ performs action a and continues like P . The parallel composition $P[A \mid B]Q$ results in an interleaving of P and Q except for the actions in $A \cap B$, which have to be performed synchronously. The processes $P \sqcap Q$ and $P \sqbox Q$ represent internal and external choice between P and Q , respectively. That means, while $P \sqcap Q$ performs an internal (τ -)action when evolving into P or into Q ,

for $P \sqbox Q$ this requires an observable action of either P or Q . For example, $(a \rightarrow P) \sqcap (b \rightarrow Q)$ performs τ in order to become either $a \rightarrow P$ or $b \rightarrow Q$. Instead, $(a \rightarrow P) \sqbox (b \rightarrow Q)$ must perform a or b and evolves into P or Q , respectively. This distinction shall be relevant for the translation of statechart diagrams below. Finally, the process $P \setminus a$ behaves like P except that all occurrences of action a are hidden.

The semantics of CSP is usually defined in terms of *traces*, *failures*, and *divergences* [14]. A trace is just a finite sequence $s \in \mathcal{A}^*$ of actions which may be observed when a process is executing. A failure (s, A) provides, in addition, the set A of actions that will be refused by the process after executing s . Divergences are traces that are followed by infinite internal computations (without any communication). We denote by $\mathcal{T}(P)$ the set of all traces of P .

Together with these semantic models come several notions of process refinement. We write $P \sqsubseteq_{\mathcal{T}} Q$ if $\mathcal{T}(Q) \subseteq \mathcal{T}(P)$, i.e., every trace of Q is also a trace of P . Analogously, $P \sqsubseteq_{\mathcal{F}} Q$ if the failures of Q are included in the failures of P , etc. In general, the idea is that Q is a refinement of P if Q is more deterministic (more completely specified) than P . These refinement relations shall be used to express consistency constraints.

D. Definition of a partial mapping

After choosing the semantic domain of CSP, we now have to define a partial mapping of models into CSP. On the one hand, this mapping must take into account all aspects leading to the consistency problem. On the other hand, the mapping should only take into account these important aspects in order to keep the later analysis feasible.

In order to be able to focus on the important aspect, we first refine the communication modeling aspect, now considering how the communication is modeled. Concerning the consistency problems, we focus on protocol statecharts and capsule statecharts and construct a table of comparison (see Figure 9), showing the differences and also similarities concerning the communication modeling aspect. We note that local behavior is only modeled within the capsule statechart and not in the protocol statechart and can thus not lead to the consistency problems. It can be derived that the mapping does not need to take into account local behavior.

Our mapping of statecharts to CSP processes is inspired by Hiemer [13]. For a precise description of our mapping, the reader is referred to [6] and [7]. In the following, we briefly summarize our results:

For each capsule statechart S , a CSP process is constructed (denoted by $\text{CSP}(S)$) parameterized over the states of the statechart. Similarly, we also translate the protocol statechart into a CSP process. Additionally, we define for each CSP process $\text{CSP}(S)$ a view process that restricts the process to the messages exchanged via a specific port p and denote this process by $V_p(\text{CSP}(S))$. Informally, this view process mirrors our decision to concentrate on messages sent over a specific port.

Diagram Modeling Aspect	Protocol Statechart	Capsule Statechart
number of participants of the communication	multi-party	single-party
kind of behavior description	global	local
direction of communication	uni-directional	bi-directional
order of messages	order of messages for participants	messages ordered for one capsule
local behavior	not modeled	modeled by local actions
distinction between synchronous and asynchronous messages	not modeled explicitly	not modeled explicitly
lifetimes of objects	not modeled	not modeled explicitly
temporal constraints	not modeled	not modeled

Fig. 9. Modeling aspects of capsule and protocol statecharts

According to the UML specification, each statechart is associated to an event queue where incoming messages are stored. However, the behavior of such an event queue is not defined. In order to be both precise and flexible about the size of buffers and their behavior, we assume that the storage and scheduling of events is the task of the connectors. Currently, connector behavior is not supported by UML-RT. For this reason, we propose to use *connector stereotypes* for selecting specific, pre-defined connector behavior. We associate every connector stereotype to a CSP process CON describing its behavior.

Given two capsules connected by two ports p_1 and p_2 via a connector associated to a connector process CON and capsule statecharts associated to the capsules named S_A and S_B , we define the semantics of this construct by

$$V_{p_1}(CSP(S_A)) \parallel [p_{1_{in}}, p_{1_{out}}] \parallel CON \parallel [p_{2_{in}}, p_{2_{out}}] \parallel V_{p_2}(CSP(S_B))$$

We denote this process with $CSP_{p_1, p_2}(S_A, CON, S_B)$.

E. Specification of semantic consistency conditions

Having described the mapping of partial models into the semantic domain CSP, we are now able to specify consistency conditions for models translated to CSP. Referring to Figure 8, for each prototypical situation we must specify the desired semantic consistency. This desired semantic consistency describes what properties the implementation should have and is deduced from the consistency requirements specified in the first step of the methodology.

Concerning the first prototypical situation, we require the behavior resulting from the execution of the components to be deadlock free. This can be formulated as a consistency condition as follows:

Condition 1 (Deadlock freeness) Two capsule statecharts S_A and S_B of two capsules connected by a connector with behavior CON via the ports p_1 and p_2 are consistent, if the induced system of CSP processes $CSP_{p_1, p_2}(S_A, CON, S_B)$ is deadlock free.

Concerning the second prototypical situation, we require that the interaction resulting from the execution of the capsule statecharts conforms to the protocol. This can be checked using a refinement notion between the CSP process of the capsule-connector-capsule construct and the CSP process of the protocol. Accordingly, we define:

Condition 2 (Protocol Consistency) Two capsules A and B connected by a connector CON via the ports p_1 and p_2 are consistent with a protocol P iff $CSP(P) \sqsubseteq_{\mathcal{F}} CSP_{p_1, p_2}(S_A, CON, S_B)$.

It is important to notice that we must base this consistency condition on the failures model of CSP because the traces model only allows the specification of safety conditions [14]. However, we want the capsules to exhibit the complete protocol behavior and not just a part of it. The traces model can only be used for expressing that some action will not occur and not that all actions indeed occur.

With respect to the development process, we already discovered that consistency problems concerning the consistency of the interaction only occur within the last step of our simplified development process. Semantic consistency checks should be performed by first checking deadlock freeness because this only involves the translation of capsule statecharts. If this condition also holds, protocol consistency should be checked by additionally translating

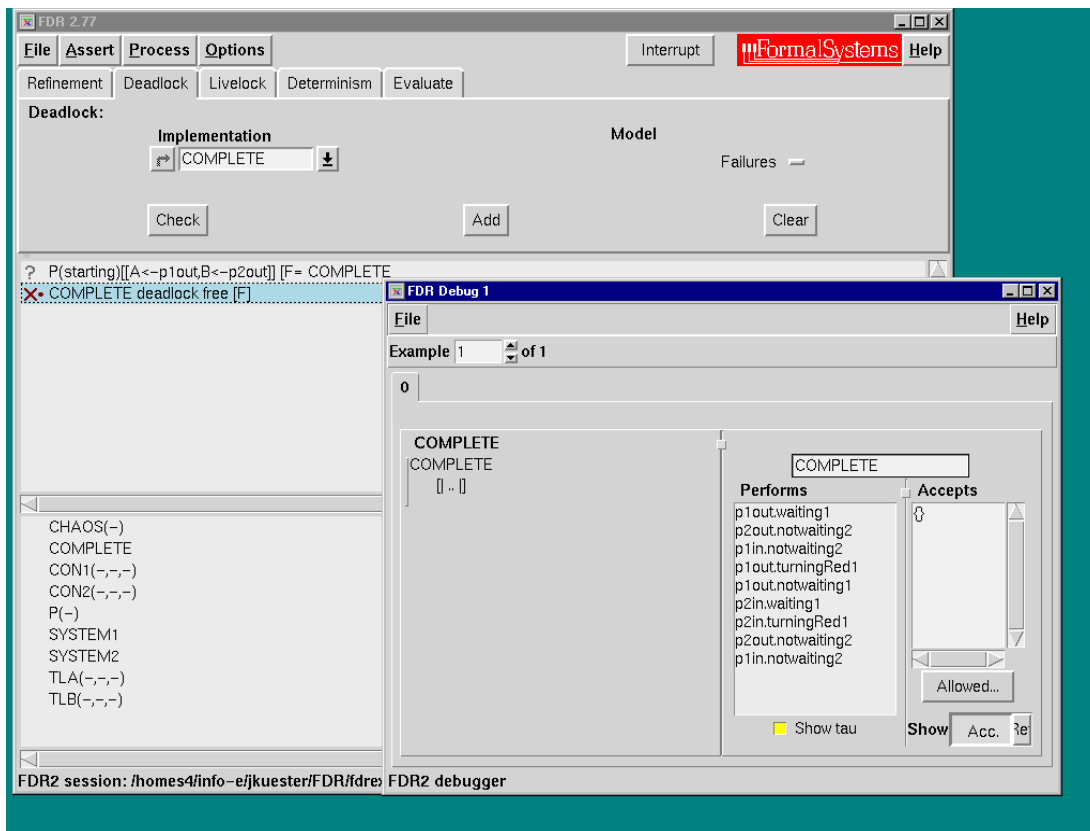


Fig. 10. The FDR tool and a trace to a deadlock

the protocol statechart.

F. Location of inconsistencies and analysis

Given a concrete UML-RT model, we can use the prototypical situations for the consistency problems to locate potential inconsistencies. This is done by finding all parts of the model where the prototypical situation arises, thereby finding all potential consistency violations.

First, syntactic consistency between the capsule statecharts and of the capsule statecharts with the protocol statechart should be checked. If syntactic consistency is ensured, semantic consistency conditions are evaluated using the model checker FDR [9] for CSP that allows the evaluation of these conditions. The output of the model checker will then be either that the model is consistent with respect to the semantic consistency conditions or it will output a counterexample (see Figure 10).

VII. CONCLUSION

In this paper, we have discussed the issue of modeling of software components with a focus on how to achieve consistency of interaction on the model level. We first discussed the issue of modeling in general and identified

that the composition of software components and their interaction can be studied on the model level if the abstraction from system components preserves all aspects that are relevant and if the meaning of models is fixed such that reasoning and analysis of models can be performed. We pointed out the importance of the choice of the modeling language and then tackled the problem of modeling components with UML, thereby discovering that UML has shortcomings concerning the consistency of models.

Therefore, we have studied the notion of consistency in detail. A model made up of submodels is called consistent if the submodels can be integrated into a single model with a well-defined semantics. The need for consistency management within modeling has been identified and it has been shown that related approaches currently do not offer a concept of consistency management for UML models. A general methodology for consistency management of UML models has been presented. Our approach is based on the idea that submodels describing common aspects of the system must have a non-contradictory semantics and should therefore be integratable into a common semantic domain. The methodology has been applied to the problem of achieving consistent interaction of software components modeled in UML-RT which is a UML profile incorporating

the notion of components and connectors.

Future work can be grouped into several directions. Concerning the methodology, this must be applied to other application domains in order to show its general applicability. With respect to the concrete steps of the methodology, the choice of the semantic domain could be supported by a catalogue of semantic domains and their relationships to modeling aspects. In order to facilitate the detection of consistency violations by the software engineer, the extension of the methodology to include a step of constructing UML models from the output of the model checker may become necessary.

Another direction of research can be seen in the improvement of the UML in order to enable the exchange of consistency conditions within a profile. This will require a more sophisticated profile mechanism and case tool support for being able to store and change consistency conditions.

Finally, concerning the modeling of components, our work should be extended to other aspects of component-based systems such as reconfiguration, yielding a more complete set of consistency conditions and thereby enabling further analysis of component-based systems on the model level.

REFERENCES

- [1] E. Boiten, H. Bowman, J. Derrick, P. Linington, and M. Steen. Viewpoint consistency in ODP. *Computer Networks*, 34(3):503–537, August 2000.
- [2] E. Boiten, H. Bowman, J. Derrick, and M. Steen. Viewpoint consistency in Z and LOTOS: A case study. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 644–664. Springer-Verlag, September 1997.
- [3] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 344–366. Springer-Verlag, New York, NY, 1997.
- [4] B. Cheng, L. Campbell, and E. Wang. Enabling automated analysis through the formalization of object-oriented modeling diagrams. In *Proceedings of IEEE International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2000.
- [5] G. Engels and L. Groenewegen. Object-oriented modeling: A roadmap. In Anthony Finkelstein, editor, *Future Of Software Engineering 2000*, pages 105–116. ACM, June 2000.
- [6] G. Engels, J. M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In Volker Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC)*, pages 186–195. ACM Press, 2001.
- [7] G. Engels, J. M. Küster, and R. Heckel. Rule-based specification of behavioral consistency based on the UML meta-model. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools., 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *LNCS*, pages 272–287. Springer, 2001.
- [8] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 84–99. Springer-Verlag, 1993.
- [9] Formal Systems Europe (Ltd). *Failures-Divergence-Refinement: FDR2 User Manual*, 1997.
- [10] P. Fradet, D. Le Métayer, and M. Périn. Consistency checking for multiple view software architectures. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 410–428. Springer-Verlag / ACM Press, 1999.
- [11] C. Ghezzi and B. A. Nuseibeh. Special Issue on Managing Inconsistency in Software Development (2). *IEEE Transactions on Software Engineering*, 25(11), November 1999.
- [12] M. Grosse-Rhode. Integrating Semantics for Object-Oriented System Models. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Proceedings of ICALP'01, LNCS 2076*, pages 40–60. Springer-Verlag, 2001.
- [13] J.-J. Hiemer. *Statecharts in CSP: Ein Prozessmodell in CSP zur Analyse von STATEMATE-Statecharts*. DrKovac Verlag, 1999.
- [14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [15] X. Li and J. Lilius. Timing analysis of UML sequence diagrams. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *LNCS*, pages 661–674. Springer, 1999.
- [16] A. Moreira and R. Clark. Combining object-oriented modeling and formal description techniques. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, pages 344 – 364. LNCS 821, Springer Verlag, 1994.
- [17] B. Nuseibeh, S. Easterbrook, and A. Russo. Making Inconsistency Respectable in Software Development. *Journal of Systems and Software*, 56(11), November 2001.
- [18] Object Modeling Group. *Unified Modeling Language Specification, version 1.4*, September 2001.
- [19] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schürr. UML + ROOM as a standard ADL? In *Proc. ICECCS'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems, Las Vegas, USA*. IEEE Computer Society Press, 1999.
- [20] B. Selic. Using UML for modeling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–262. Springer Verlag, 1998.
- [21] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.