

Management of UML Clusters

Peggy Schmidt and Bernhard Thalheim

Christian-Albrechts-University Kiel, Computer Science Institute, 24098 Kiel,
Germany

pescthalheim@is.informatik.uni-kiel.de

Abstract. Software engineering uses UML diagrams as a standard technique for specification and development of software. Various UML diagrams are used for specification of different aspects of the application. Their interpretation, extension, revision and integration becomes awful difficult if developers use the full freedom of UML, apply their own semantics and do not agree on common parts. We propose an approach that limits this freedom to the necessary extent. Developers have the full freedom on parts of the specification that is independent from others and are committed to fulfill contracts on parts of the specification that is also used by other developers.

Due to a lack of semantics the integration of various UML diagrams is often left to the intuition of software engineers, which bears the risk of UML-based software development becoming error-prone. In this paper we propose the use of Abstract State Machines (ASMs) as a means to support the integration of UML diagrams by means of invertible translations of *UML clusters*, i.e. sets of UML diagrams together with constraints defined on them, into easily understandable ASM specifications. In doing so, the rigorous semantics of ASMs induces an unambiguous semantics for the UML clusters. These translations themselves can be formalised by ASM specifications thereby automating the translation process. Furthermore, the evolution of UML clusters is guarded by *contracts*, which can again be specified by ASMs.

1 Introduction

Research has been conducted in software engineering for more than forty years. Ideas which have been studied include support for different levels of abstraction, information hiding, and reasoning with local computations. The goal of software engineering is to create high-quality software. Enterprises are becoming increasingly complex in the information age. To realize the building of complex information systems it is essential to resolve such problems as high level specification and target planning at the concept level. Software engineering has produced an enormous amount of notation and methodology that aims to handle the software process.

1.1 Software Engineering and Software Specification

Software engineering is thus the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation

required to develop, operate, and maintain them [Boe76]. UML diagrams are ubiquitous in software engineering, forming the cornerstones of modelling techniques. UML intentionally leaves open semantics and formalisations of UML diagrams. This variety of understanding makes the management of UML a mental challenge for large projects and in the context of team development. The pUML (precise UML) initiative [pUM07] tries to add some mathematical rigor to UML but is not yet supported by tools.

Problem 1. Adding mathematical rigor to representations: Each work product in software engineering should have an operational semantics. This semantics should allow the development of a (small) logical theory on the basis of which properties of specifications can be proven, validated or verified.

Software products are typically developed in large team whose members are locally developing products. The completion of the development task and the quality [ISO01] is treated in very different ways. The final goal of the software development process is however an integrated and well-behaved product.

Problem 2. Integrated development of different representations: UML diagrams are used to specify different views of the same software. They must be used consistently in an integrated form. Their integration must be made explicit.

Software engineering has focused in the past mainly on development processes such as requirements engineering, conceptual system specification, implementation, maintenance, testing, introduction, and deployment. Whereas the latter processes are interwoven the first three processes are more or less sequential. Each development step changes one or two of the development products and leaves the others out.

Problem 3. Flexible change management: Any change to one of the development products must take into consideration the changes required by the current changes in other software products.

Software engineering products are documents such as UML diagrams. They are revised and changed during the development process. We need a mechanism that supports the evolution of software specification products and reasoning their properties quality. UML provides a way of communicating between developer and user, and is well accepted in research as well as in industry. UML collects a federation of different models with different views and scope for the product. UML issues a rich set of pictorial and graphical notations. Currently UML is comprised of miscellaneous notations with no formal meaning. The problems of UML include a large number of diagram types and a consequent underspecification of their semantics. There are, however, main differences in the semantics of these diagrams and in the way these diagrams are used.

Problem 4. Evolution of different representations: Changes within any UML document, must either be refinements of previous diagrams or explicit revisions of such diagrams. These changes must be enforced for other diagrams as well whenever those are concerned too.

We claim that these four problems can be solved on the basis of abstract state machines (*ASM*). *ASM* [BS03] were introduced as a general mathematical framework for systems specification and implementation. This framework supports rough, sketchy specification as well as detailed, fine-grained specification. Both types of specifications can be seen as a machine. The behaviour of these machines can be simulated.

The different specifications vary in their level of detail. We want, however, that one specification can be considered to be a refinement [Bör03] or a revision of the other one. A refinement is generally defined (1) by a structural scope of interest for both machines, (2) by an equivalence relation on states of both structural scopes of interest, (3) by a behavioural scope of interest called a computational segment, and (4) by a partial equivalence relation between the behavioural scope of interest. We require that these equivalence relations allow any behaviour of one machine to be seen or understood as a behaviour of the other machine.

1.2 Requirements for Management of UML Specifications

UML diagrams are typically developed either by hand using drawing tools or with support of UML tools such as Rational Rose. The interpretation of the diagrams is often fuzzy and may be hidden within the assumptions of the tools. The UML standard has left this interpretation to the developers and implementers. Therefore we must provide a very flexible framework that allows interpretations to be injected at a later stage. These diagrams are intertwined and depend on each other based on agreement. We thus must cope with the *coexistence* of these diagrams. They allow a specific view on the system that is under development.

The development process has typically four dimensions. The *integration dimension* of a set of diagrams is concerned with the incorporation of several diagrams depending on the overlap and viewpoints supported by the diagram. The *abstraction layer dimension* considers relations among different abstract views on the product, e.g., the realisation of the analysis diagrams by the system specification. The *evolution dimension* adds complexity to the development process by modifying existing diagrams to new diagrams or initiating the development of new diagrams. For example, first a use case diagram is specified and second the state-chart diagram under consideration of the use case diagram. The *collaboration dimension* must take care on the steps that developers may apply while working in a development team. It is not surprising that only the first dimension is partially supported by tools. UML tools are harmed by the lack of a general formal framework of each of the UML diagrams and by the resulting lacking support for integration of diagrams. The integration and the abstraction layer dimensions have not got a satisfying treatment inside UML. The survey [EB04] covers around 4 dozen papers. Constraint examples considered are typically rather trivial. The main kind of constraints are constraints on well-formedness and on existence. Moreover, very few research has been carried out on the evolution dimension and the collaboration dimension.

At the same time a large body of knowledge is available for formal foundation of each of the UML diagrams. We prefer the *ASM* research that has

resulted in a formal basis of the most important UML diagrams such as class diagrams [Obe01], statecharts [BCR00b], use case diagrams [BGS⁺03], and activity diagrams [BCR00a]. We thus may assume that each of the diagrams can be embedded into their ASM. The consistency of these ASM interpretations has not yet got a satisfying solution.

1.3 Organisation of the Paper

Section 2 introduces UML diagram clusters and contracts that allow to manage UML diagram clusters. The explicit formulation of clusters and of contracts is novel and has not been considered in the past. Section 3 describes the support of UML clusters by an *ASM* specifications. We give examples for the transformation UML-Diagrams into *ASM* specifications and for the check of consistency between UML clusters in Section 3 and 4. Section 4 demonstrates the potential of our method by applying it to collaborative development of software within a team of developers. Section 5 summarizes the paper.

2 UML Diagram Clusters

The software-development process yields a partially ordered set of UML models that should be consistent. Their semantics should be precisely defined. At the same time it must however defined in a very flexible way and is going to be refined through UML diagram set transformation. The problem thus arises of how to understand and how to check consistency among diagrams and between different versions.

2.1 Algebraic UML Systems for Kinds of UML Diagrams

Signatures \mathbb{S} define the kind of UML diagrams. Typically, signatures describe the base canon that is used within the abstract syntax. Most signatures are given by the specification of the sorts or domains, of the (dynamic) predicates and functions and by formal grammars for inductive construction of complex elements. These grammatical rules may lead to syntactically different but semantically equivalent expressions. Therefore, we use an equivalence relation for expressing the same meaning. Typically, equivalences can be defined on the basis of an equality system.

A UML diagram may be defined as a labelled graph $\mathcal{G}_{\mathbb{S}}$ of certain signature \mathbb{S} with a set of integrity constraints or well-formedness rules. These rules are expressed either in the OCL well-formedness rules which extend the syntactic rules given in the meta-language or in first-order predicate logic. The nodes correspond to main elements of signature \mathbb{S} and edges correspond to their relations. Labels are used for providing additional information on the nodes and edges.

For instance, a statechart diagram is given by a set of states and their transitions. A initial/ final state has at least one outgoing / incoming transition and no incoming / outgoing transition. Attribute and operation identifiers in class

diagrams may required to be unique. We may also use general constraints such as the *unique name assumption* that requires names to be unique within a diagram.

Such conditions can be considered to be constraints that specify well-formedness $\Sigma_{\mathbb{S}}^{\text{WellFormed}}$ of diagrams. This set also includes any equivalence of interest for the given kind of diagrams.

In general, we may weaken requirements to well-formedness for the intermediate steps in a development process. For instance, the assignment of *public*, *private* and the specification of operations can be delayed. Additionally, UML intentionally leaves open semantics of diagrams. For instance, a number the options for semantical interpretation of statecharts is discussed in [BCR00b]. At least four different decisions for semantics of UML statecharts must be made before a statechart diagram interpretation can be generated: (a) event handling (whether an event is considered to be a releasable and dispatched event); (b) concurrency treatment for transitions in compound states; (c) environment for generation of state completion; (d) selection and handling of transitions.

A similar variety of interpretations can be observed for class diagrams, activity diagrams, and interaction diagrams. The clarification of class diagram semantics may either be based on semantics of object-oriented databases [ST93, Obe01] or on proposals made by the pUML (precise UML) initiative [pUM07]. The interpretation of use diagrams we use is based on the SiteLang language [DT01] that has been developed for specification of user behaviour within information-intensive web systems.

Since interpretations do not have a Church-Rosser property and one choice for interpretation of one construct may restrict the application of an interpretation of another construct we restrict our consideration to the **semantics integration assumption**: *Given a kind of UML diagrams defined over a signature \mathbb{S} . Restrictions and equivalences applied to this kind of diagrams are not contradicting.* If this assumption is applicable then we may assume that all diagrams of the given kind may have a strongest and a weakest interpretation (final and free interpretation) defined by the conjunction or the disjunction of all restrictions, respectively. The interpretations form a lattice of hierarchically ordered equationally partial heterogeneous structures or algebras [Mal70, Rei84].

This approach may be combined with the theory of institutions and the CASL approaches [BST02] to algebraic UML systems. Institutions provide a set of signatures, a set of formulas defined on these signatures and a set of model classes for the signatures. Model classes are related by a satisfaction relation that define whether a model M of some signature \mathcal{S} satisfies a formula α from $\mathcal{L}_{\mathcal{S}}$.

We assume now that the signature is fixed for one kind of UML diagrams. The different interpretations possible form a lattice $\mathcal{L}(\Sigma_{\mathbb{S}}^{\text{WellFormed}})$ of subsets of $\Sigma_{\mathbb{S}}^{\text{WellFormed}}$ with the set $\Sigma_{\mathbb{S}}^{\text{WellFormed}}$ as the largest element and a subset of $\Sigma_{\mathbb{S}}^{\text{WellFormed}}$ as the smallest element that is considered the lowest standard for well-formedness of diagrams. Additionally, modification operations must be specified for UML diagrams. These operations may be considered as specialisations of typical graph and schema editing operations known for database schemata

[Tha00]. Some of these operations can contingent others, i.e. can be used to enhance the other operations in such a way that constraints are becoming fulfilled if these operations are applied. Some of these operations can also be compensating operation that allow to undo other operations.

An algebraic UML system $(\{\mathcal{G}_{\mathbb{S}}\}, \mathcal{L}(\Sigma_{\mathbb{S}}^{\text{WellFormed}}), \mathcal{O}_{\mathbb{S}})$ of signature \mathbb{S} is given by all labelled graphs $\{\mathcal{G}_{\mathbb{S}}\}$ on signature \mathbb{S} , a lattice $\mathcal{L}(\Sigma_{\mathbb{S}}^{\text{WellFormed}})$ of sets of constraints, and by modification and development operations $\mathcal{O}_{\mathbb{S}}$ that can be used for the development of diagrams of signature \mathbb{S} .

2.2 Contracted Development of UML Diagrams

Consistency of sets can be specified through a set of logical formulas that are specified in an appropriate language. For instance, database applications use integrity constraints which specify which states of the database are desirable and which states are forbidden. UML diagrams can be considered as a set of abstract objects. The meaning of these objects is the illustration of different viewpoints and properties of a program. The UML meta-language provides a framework for description of objects that are represented by a UML diagram. These objects typically have complex structures and are well-formed according to the requirements of the UML standard [HKKR05]. A set of UML diagrams describe an application. Therefore, UML diagrams must also obey consistency constraints. Typically, these consistency constraints are not explicitly specified. The explicit specification and the sophisticated treatment of these constraints contributes to the solution of the four problems discussed in Section 1.1. We base our approach on a four-layer treatment in contracted development:

1. Declaration of constraints that are applied to a singleton diagram or to sets of coexisting diagrams;
2. Description of enforcement mechanisms (when must the constraint checked, how the constraint is checked, what to do if the constraint is violated, what mechanism can be used to trigger the constraint) that support constraint validity during development, change, and evolution of UML diagrams;
3. Description of change and evolution steps that can be applied for refinement or modification of sets of UML diagrams based on scopes of constraints and operational use of constraints;
4. Support by tools or workbenches that maintain validity of constraints.

The third layer may also consider the development of UML diagrams within development teams. In this case, team members are supported by approaches to collaboration, e.g. explicit services and exchange frames [ST07].

A contract ζ^{Contract} consists of a declaration of constraints, of a description of the enforcement mechanism and of a prescription of modification steps that transform a consistent set of diagrams into a consistent set of diagrams.

A contract may include obligations, permissions and sanctions. Therefore,

- contracts declare correctness of a set of diagrams, separate exceptional states from normal states for these sets, and forbid meaningless sets of diagrams,

- contracts enable the direct manipulation of the set of diagrams as transparently as possible and offer the required feedback in the case of invalidation of constraints based on echo back, visualisation of implications, on deferred validation, instant projection and hypothetical compilation, and
- contracts consider mechanisms that address the long term integrity of diagram sets by forecasting confirmation, by anticipating changes made in a team, by providing a mechanism for adjusting and confirming correctness, and by specifying diagnostic queries for inspection of diagram sets.

A typical constraint on a set of diagrams is the existence constraint

$$EC_1^{states(SC,CT)} : \text{StateChart}(\text{States}) \subseteq \text{ClassDiagram}(\pi_X(\text{RulingClass}))$$

where *RulingClass* is a class defined in the class diagram and $\pi_X(Y)$ is the projection function. The constraint requires that states in statecharts must be defined by attributes of the ruling class in a class diagram. Changes to states are thus restricted. We declare

$$\text{State}' \notin \text{ClassDiagram}(\pi_X(\text{RulingClass})) \longrightarrow \\ F \text{ modify}(\text{StateChart}(\text{State}, \text{State}')),$$

$$O \text{ cascade}(\text{modify}(\text{ClassDiagram}(\text{RulingClass}, X)), \text{modify}(\text{StateChart}(\text{State})))$$

where `modify` denotes any diagram modification operation applicable to the signature of the diagram, `cascade` denotes an obligation to apply a second action if the first action has been completed and *F*, *P*, *O* are the deontic operators forbid, permit, oblige. Furthermore, we assume

$$\text{do}(A_1, \text{modify}(\text{ClassDiagram}(\text{RulingClass}, X))) \longrightarrow \\ \text{do}(\text{notify}(A_2, \text{modify}(\text{ClassDiagram}(\text{RulingClass}, X))))$$

for any two agents A_1 and A_2 where A_1 the right to modify class diagrams and A_2 has the right to read the statechart.

For instance, in the running example we require that the states *IsBorrowed*, *IsReturned* are definable for the class *Book*, i.e.

$$\text{StateChart}(\{IsBorrowed, IsReturned\}) \subseteq \text{ClassDiagram}(\pi_{LendingState}(\text{Book})) .$$

Contracts typically follow a number of norms that are given by laws, regulations or agreements among the parties involved. Our treatment generalise this understanding. Parties involved into a contract are either singleton diagrams or team members involved in a development project. A contract may be extended by the following information: roles of the parties that are involved; relationships between contracting parties; begin and end of contract; the status of contracts; a contract monitoring facility that performs checking of the fulfillment of obligations and compliance monitoring; a contract notification component that sends various contract notifications to the roles involved in contract management; other components and facilities to support contract negotiations, enforcement and also dynamic configurations of the system to reflect new business rules and structures.

2.3 Coexistence of UML Diagrams and UML Clusters

During UML-based system specification a number of UML diagrams is used for description of different viewpoints in different levels of detail on the application. UML does not provide mechanisms for support of coexistence of diagrams beyond

the UML meta-model, the Object Constraints Language OCL [HKKR05] and rules developed within the precise UML initiative. Coexistence of diagrams must be handled within all four dimensions: integration, abstraction layer, evolution and collaboration.

The development of UML diagrams is performed by agent (or actors) that obtain rights to apply (do) modification and development operations and can be obliged to apply these operations during their work. Rights are ordered. For instance, an agent that can modify diagrams have typically the right to read these diagrams.

Beside well-formedness we might also consider any kind of constraint over diagrams of signature \mathbb{S} . For instance, hierarchies in class diagrams must be acyclic. States in statecharts must uniquely determine the node in the statechart. Therefore, we define a **diagram type** $\mathcal{T}_{\mathbb{S}} = (\mathbb{S}, \Sigma_{\mathbb{S}})$ by the signature of the diagram and by constraints $\Sigma_{\mathbb{S}} \in \mathcal{L}(\Sigma_{\mathbb{S}}^{\text{WellFormed}})$ applicable to all diagrams defined over this type .

Constraints in UML clusters can be categorised into

- existence constraints** $\mathbb{S}_1[E] \subseteq \text{Exp}(\mathbb{S}_2)$ that bind the utilisation of one element set E in a diagram of signature \mathbb{S}_1 to the existence of this element or of an expression that declares this element within a diagram of signature \mathbb{S}_2 ,
- visibility constraints** $E \subseteq \text{Public}(\text{Exp}(\mathbb{S}_2))$ for $E \subseteq \text{Exp}(\mathbb{S}_1)$ that require that expressions used in one diagrams must be visible on the other diagram,
- cardinality constraints** $\text{card}_{\mathbb{S}_1} = (m_1, n_1) \longrightarrow \text{card}_{\mathbb{S}_2} = (m_2, n_2)$ that restrict the multiplicity of elements in one diagram is bind by the multiplicity of elements in another diagram,
- refinement constraints** $e = E$ for elements e in a diagram of signature \mathbb{S}_1 and expressions $E \in \text{Exp}(\mathbb{S}_2)$ that restrict the refinement of diagrams of signature \mathbb{S}_1 by diagrams of signature \mathbb{S}_2 and
- evolution constraints** that specify the consistency of different versions of the same application.

For instance, a link between a sender and receiver in a sequence diagram must be based on the existence of a corresponding association in the class diagram. All elements in a sequence diagram require that the corresponding classes, attributes, operations and references are visible in the class diagram. Since messages in sequence diagrams can initiate creation or deletion of object the cardinality of these object sets must be consistent with the cardinality constraints specified for the class diagram. Use cases in a use case diagram can be refined by an entire activity diagram.

A UML cluster type $\mathcal{CT} = (\mathcal{T}_{\mathbb{S}_1}, \dots, \mathcal{T}_{\mathbb{S}_n}, \Sigma_{\mathbb{S}_1, \dots, \mathbb{S}_n})$ is given by UML types $\mathcal{T}_{\mathbb{S}_i}$ defined on a set $\mathbb{S}_1, \dots, \mathbb{S}_n$ of signatures and a set $\Sigma_{\mathbb{S}_1, \dots, \mathbb{S}_n}$ of constraints on these signatures. A UML cluster \mathcal{C} on a cluster type \mathcal{CT} consists of UML diagrams $(\mathcal{D}_1, \dots, \mathcal{D}_n)$ of type $\mathcal{T}_{\mathbb{S}_i}$ that obey $\Sigma_{\mathbb{S}_1, \dots, \mathbb{S}_n}$. The contract on \mathcal{CT} thus consists of the constraints $\Sigma_{\mathbb{S}_1} \cup \dots \cup \Sigma_{\mathbb{S}_n} \cup \Sigma_{\mathbb{S}_1, \dots, \mathbb{S}_n}$, a description of the enforcement mechanisms for any operation that can be used for modification of one UML diagram, and a set of consistent evolution transformations.

The evolution steps may be very complex. We may either use a transaction approach that accept only those modification step sequences which are correct or may develop modification operations which are correct. We prefer the second approach and aim in development of atomic steps. These steps must obey the contract. Therefore we also need compensation and contingent operations that compensate the operation or that continue the operation in the case that the step has led to a cluster which is not consistent. The final result of evolution steps is typically a UML cluster. Currently, all approaches prioritise one diagram signature and consider the corresponding diagram to be the final product supported by the other diagrams.

3 ASM-Based UML Clusters

3.1 ASM Basis for UML Clusters

The coexistence of UML diagrams can be either supported on the basis of algorithms that check contracts, e.g., [Tsi00], or map the consistency to certain logics or check consistency on the basis of operational semantics. Most approaches develop algorithms for certain classes of constraints. Any new class of constraints must thus be supported by new algorithms. Some few approaches map constraints to certain logics such as description logics, e.g. [SMSJ03]. Unfortunately [Tha00], description logics are already insufficient for handling of cardinality constraints. We prefer operational semantics. We map UML diagrams to an ASM based on the mappings that have already been given in [BGS⁺03, BCR00b, BCR00a, Obe01]. This mapping achieves an integration of all diagrams. We therefore use an operational consistency paradigm based on the requirement that a UML cluster has a common semantic interpretation.

For instance, the transformation of class diagrams to ASM is based on the theory of object-oriented databases [ST93, ST99] and [Obe01]. We use the layering in Section 2.3. Let us assume that a number of static domains such as *Name*, *AttrName*, *OpName*, *AssocName*, ... are given. The signature of class diagrams is given by

$\mathcal{ASM}_{\text{ClassTypeSystem}}$

$$\begin{aligned} \mathcal{T}^{\text{CT}} &= \{\tau_1, \dots, \tau_{n_T}\}, & \mathcal{A}^{\text{CT}} &= \{A_1, \dots, A_{n_A}\}, & \mathcal{O}^{\text{CT}} &= \{o_1, \dots, o_{n_O}\} & \mathcal{E}^{\text{CT}} &= \{e_1, \dots, e_{n_E}\} \\ \text{name} &: \mathcal{T}^{\text{CT}} \rightarrow \text{Name} \\ \text{attr}^T &: \mathcal{T}^{\text{CT}} \times \mathcal{A}^{\text{CT}} \rightarrow \text{AttrName} \times \text{Visibility} \times \text{AttrDataType} \\ \text{operation}^{\mathcal{T}^{\text{CT}}} &: \mathcal{T}^{\text{CT}} \times \mathcal{O}^{\text{CT}} \rightarrow \text{OpName} \times \text{Visibility} \times \text{ParamList} \times \text{MethDataType} \\ \text{association}^{\mathcal{T}^{\text{CT}}} &: \mathcal{T}^{\text{CT}} \times \mathcal{T}^{\text{CT}} \times \mathcal{E}^{\text{CT}} \rightarrow \text{AssocName} \times \text{AssocKind} \\ \text{association_attr_source}^{\mathcal{T}^{\text{CT}}} &: \mathcal{T}^{\text{CT}} \times \mathcal{T}^{\text{CT}} \times \mathcal{E}^{\text{CT}} \rightarrow \text{Role} \times \text{Card} \times \text{Visibility} \\ \text{association_attr_source}^{\mathcal{T}^{\text{CT}}} &: \mathcal{T}^{\text{CT}} \times \mathcal{T}^{\text{CT}} \times \mathcal{E}^{\text{CT}} \rightarrow \text{Role} \times \text{Card} \times \text{Visibility} \\ &\dots \end{aligned}$$

The entire translation process results in an $\mathcal{ASM}_C^{\text{Spec}}$ specification of the cluster. This specification also reflects all constraints in the cluster. For instance, we use constraints expressing well-formedness of diagrams. Let us consider a constraint which requires that names of associations uniquely determine the kind of the association (normal, generalisation, ...)

constraint $\mathbf{WF}_{FD1}^{CT} : \forall \tau_1 \forall \tau_2 \in \mathcal{T}^{CT} \forall \zeta_1 \forall \zeta_2 \in \mathcal{E}^{CT}$
 $(\text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_1) \neq \mathbf{undef} \wedge \text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_2) \neq \mathbf{undef}$
 $\text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_1) = (an_1, ak_1) \wedge \text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_2) = (an_2, ak_2)$
 $\wedge an_1 = an_2) \longrightarrow ak_1 = ak_2$.

We notice that a number of different transformation styles can be applied. We used for the example above an object-preserving transformation that concentrates the translation to nodes and edges of the graph. Instead we might use full unnesting that generates a function for each component of the graph. Another style uses objects such as attributes, operations, etc. on their own and associates them with the corresponding class through a function. We use the unique name assumption and assign a unique namespace to each diagram.

Constraints can be transformed to ASM rules, e.g. the constraint above to

$\mathbf{WF}_{FD1}^{CT} =$
foreach $\tau_1, \tau_2 \in \mathcal{T}^{CT}, \zeta_1, \zeta_2 \in \mathcal{E}^{CT}$ **do**
let $(an_1, ak_1) = \text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_1)$ **in**
let $(an_2, ak_2) = \text{association}^{\mathcal{T}^{CT}}(\tau_1, \tau_2, \zeta_2)$ **in**
if $an_1 = an_2 \wedge an_1 \neq \mathbf{undef} \wedge ak_1 \neq \mathbf{undef} \wedge ak_2 \neq \mathbf{undef} \wedge ak_1 \neq ak_2$ **then**
 $\text{reaction}(\zeta_1, \zeta_2) := \text{contract_reaction}(\mathbf{WF}_{FD1})$

where we assume that `contract_reaction` is a function set on initialisation.

3.2 ASM-Based Contract Management

An $\mathcal{ASM}_C^{\text{Spec}}$ specification is enhanced by assumptions $\mathcal{ASM}^{\text{Assum}}$ and guarantees $\mathcal{ASM}^{\text{Guaran}}$, i.e., $\mathcal{ASM}^{\text{Contract}} = (\mathcal{ASM}^{\text{Assum}}, \mathcal{ASM}^{\text{Guaran}})$. Guarantees can be seen as the smallest common divisor. They are going to be supported by the environment $\mathcal{ASM}_{\text{Environment}}^{\text{Spec}}$ if it behaves according to \mathcal{ASM}_A . Assumptions may also represent requirements of $\mathcal{ASM}_C^{\text{Spec}}$ to the environment. We assume that the assumptions imply the guarantees, i.e., any consistent with $\mathcal{ASM}^{\text{Assum}}$ specification obeys the guarantees.

The abstract state machines $\mathcal{ASM}^{\text{Contract}}$ handles the fulfillment of contracts. The generation process of these machines is demonstrated in Section 4.

The management of contracts is based on three steps [KN02]: registration, contract negotiation and contract execution.

Registration Two agents are involved into the registration phase and use the role of acting addressee and the role of reacting counter-party. They identify their need to be engaged in a change of $\mathcal{ASM}^{\text{Spec}}$ under the supervision of the contract manager ($\mathcal{ASM}^{\text{Contract}}$). Within the next step they may agree in principle on issues or open a negotiation. These agreements will determine the type of service required from the contract manager. The purpose of the contract will be negotiated in the following phase. The type of service is expressed as a **Contract Template** and put forward by the authority to the two contracting agents.

Contract negotiation: This step manages the domain-specific content of the contract following the template agreed upon in the registration phase. Issues determined to be important in the registration phase can be negotiated, for

example, for a decomposition step applied to a class and resulting in several classes within a UML class diagram. In general, a contract specifies the collaboration of agents whenever changes applied to $\mathcal{ASM}^{\text{Spec}}$ specifications (e.g., $\mathcal{ASM}^{\text{ClstrT}}$, $\mathcal{ASM}^{\text{Clstr}}$, or $\mathcal{ASM}^{\text{Prgrm}}$). We distinguish between *obligations* for the acting and the reacting agent, *permissions* given by the reacting agent to the acting agent, and *sanctions* applied to the $\mathcal{ASM}^{\text{Contract}}$ to the acting agent or to the reacting agent.

Contract execution: The fully negotiated contract is executed by the three agents under the supervision of the $\mathcal{ASM}^{\text{Contract}}$. The bound contract contains declarations of obligations, permissions and sanctions of each party following the template used. These declarations will lead to the execution of the contract.

Contracts describe constraints that the $\mathcal{ASM}^{\text{Spec}}$ must satisfy before using the service as well as the constraints that are guaranteed by the service when used. The **Activation** defines preconditions for obligations, permissions or sanctions. **Finalization** assumes that the **activation** is true. If the **activation** and **finalization** conditions are met, then the service **permissions** must be preserved.

A development contract may contain additional constraints. For example, the following $\mathcal{ASM}^{\text{Contract}}$ specifies that all methods in a statechart of a cluster have to exist in the class diagram of this cluster, i.e. the existence constraint

$$\text{EC}_1^{\text{meth}(SC,CT)}: \text{StateChart}(\text{Methods}) \subseteq \text{ClassDiagram}(\text{Methods})$$

is transformed to the generalised rule

$$\text{INTEGRITYCONSTRAINTTEST} \Leftrightarrow \mathcal{O}^{\text{SC}} \subseteq \mathcal{O}^{\text{CT}} \wedge \{\delta^{\tau_{\text{class diagram}}}, \delta^{\tau_{\text{statechart}}}\} \subseteq \mathcal{C}$$

and to the rule

$$\begin{aligned} \text{EC}_1^{\text{meth}(SC,CT)} = & \\ & \text{foreach } o \in \mathcal{O}^{\text{SC}} \text{ do} \\ & \text{if } o \notin \mathcal{O}^{\text{CT}} \text{ then} \\ & \quad \text{reaction} := \text{contract_reaction}(\text{EC}_1^{\text{meth}(SC,CT)}) \end{aligned}$$

3.3 Supporting Management by Contract Templates

The contract specifies the coherence within and between UML diagram clusters and supports propagation of modifications throughout the cluster. The contract states which constraints remain to be valid after a modification. Contracts may include specific styles and pattern for the treatment of modifications. For instance, contracts may be based on the ACID paradigm, i.e. any modification set is either completely applied to the cluster or rolled back, can be applied in isolation from other modification sets and is then persistent for the cluster. If contracts can generically be defined then **contract templates** can be defined. A contract template is a contract with parameters for the UML diagrams. The template is instantiated to the machine $\mathcal{ASM}^{\text{Contract}}$ that manages contracts.

4 Development of Clusters and Contracts

We may easily extend the approach to cope with several clusters. In this case we have to solve two problems at the same time:

- coexistence of diagrams within one cluster based on the cluster constraints and
- collaboration of developers that develop diagrams within different clusters.

The result of a collaboration should be a cluster that is commonly agreed between the agents. Diagrams should be mapped to an ASM^{Prgrm} specification. Any modification can only be applied to a diagram in the cluster if the contract is preserved by the ASM^{Prgrm} .

We use an $ASM^{Contract}$ for management of deployment of the ASM^{Clstr} and ASM^{Prgrm} specifications based on three services: generating a ASM^{Prgrm} specification from a ASM^{Clstr} , decomposition of classes inside an ASM^{Prgrm} , and collaboration of two UML diagram clusters.

4.1 Example: Library Support System

We illustrate the approach on the basis of a small example for a library support system. We shall use this example to demonstrate the power of our approach.

In Figure 1 two UML diagram clusters C_1 and C_2 are depicted. The first one uses a use case and a class diagram. The second one considers a start chart and a view on the class diagram of the first cluster. The use cases in Figure 1 are *borrow* and *return* a book from the library. The class diagram in the first cluster C_1 uses the classes *Book*, the *Person*, *Log4BorrowingABook*. The *statechart* diagram models the different states. In our example one can borrow a book only if the book is available (*isReturned*).

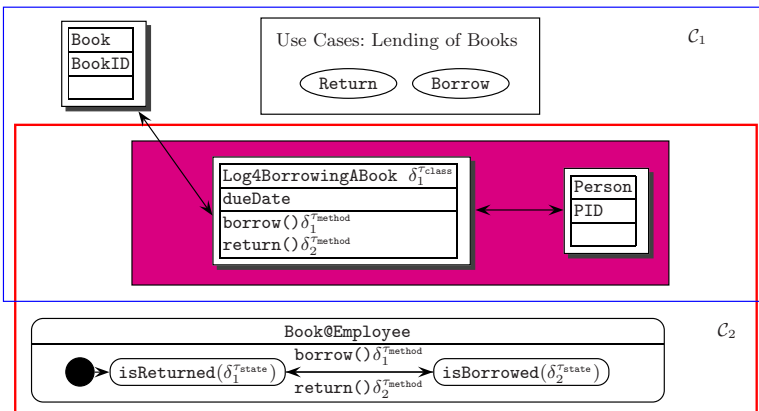


Fig. 1. Two diagram clusters developed for the *Library* application

The UML clusters developed so far can be transformed into an ASM_{C_1} and ASM_{C_2} . The transformation depends on the transformation style. For instance, one transformation style requires that each method of a class is transformed into a dynamic function. The translation style we used above transforms methods into domains.

The transformation of UML diagrams to programs or ASM can be based on translation profiles or styles similar to those developed by diagramming techniques for databases [Tha00]. The transformation result of cluster C_1 may be based on style A:

SIGNATURE OF $ASM^{Program, styleA}$

```

Log4BorrowingABookID = {1, ..., n}
...
Method = {borrow, return}
LendingState = {IsBorrowed, IsReturned}
book : Log4BorrowingABookID → BookID
person : Log4BorrowingABookID → PID
dueDate : Log4BorrowingABookID → Date
method_m : Log4BorrowingABookID → String

```

MAINRULE

```

for each  $o \in BorrowedBook$  do
  if  $method(o) == borrowBook$  then
    INITIALIZE()
  ...

```

We may also apply another style style B. It results in the following:

SIGNATURE OF $ASM^{Program, styleB}$

```

...
log4BorrowingABook: Log4BorrowingABookID × BookID × PID
                    × DueDate → Bool

```

MAINRULE

```

for each  $o \in Log4BorrowingABook$  do
  ...

```

In the current example style A is used for both clusters since the contract requires that the style for transformation must be fixed. We get the ASM in Figure 2.

4.2 Collaborative Development and Collaboration Contracts

The collaboration contract $\zeta_{C_1, C_2}^{Contract}$ is based on an equality system \mathfrak{E}_{C_1, C_2} that relates expressions in C_1 to expressions in C_2 . A simple case of such equality systems all expressions are basic elements of the cluster. In this case, we may introduce directed equalities. These equalities propose a preference which of the notions is going to be used for the integrated cluster C_{12} . At the same time, we can use inequalities for those basic elements that cannot be related to each other. The practicality of equality logics for schema integration has already been shown for database schemata in [Ves05]. It can be enhanced by cooperating views [FRT05, Tha00] which are used for database collaboration.

```

SIGNATURE OF  $ASM_{aM}^{Program, styleA}$ 
student : Log4BorrowBookByStudentID  $\rightarrow int$ 
book : Log4BorrowBookByStudentID  $\rightarrow int$ 
dueDate : Log4BorrowBookByStudentID  $\rightarrow Date$ 

MAINRULE
foreach  $o \in Log4BorrBookStud$  do
  if method( $o$ ) == borrow() then
    INITIALIZE()
  if method( $o$ ) == borrow() then
    // Code is generated from the state-chart
1   if Actor( $o$ ) == Student then
    // precondition
2   if ctl_state( $o$ ) == undef  $\vee$  ctl_state( $o$ ) == isReturned then
3     let  $o = \text{new} (Log4BorrBookStud())$ 
4     book( $o$ ) = param_book
5     Student( $o$ ) = param_Student
6     Duedate( $o$ ) = param_Duedate
    // postcondition
7     ctl_state( $o$ ) := isBorrowed
8     ctl_stateIsBorrowed( $o$ ) := isNotOD
  if method( $o$ ) == return then
    if Actor == Student then
    // precondition
9     if ctl_state( $o$ ) == isBorrowed then
10    return() // postcondition
11    ctl_state( $o$ ) := isReturned
  if Date > DueDate( $o$ ) then
    // precondition
12    if ctl_state( $o$ ) == isNotOD then
13    ctl_stateIsBorrowed( $o$ ) := isOD

```

Fig. 2. The ASM specification after generation

The equality system $\mathfrak{E}_{\mathcal{C}_1, \mathcal{C}_2}$ is now the basis for an extended contract. The contracts of the clusters $\mathcal{C}_1, \mathcal{C}_2$ can be enhanced by the equality system, the agreements on enforcement for equalities provided by $\mathfrak{E}_{\mathcal{C}_1, \mathcal{C}_2}$ and the modifications that can be applied to \mathcal{C}_1 or \mathcal{C}_2 . This contract part is called **collaboration contract** $\zeta_{\mathcal{C}_1, \mathcal{C}_2}^{\text{Contract}}$. It defines what each agent in the development process of their clusters $\mathcal{C}_1, \mathcal{C}_2$ should do.

The collaboration contract can also be represented by a contract of the union of the two clusters. UML clusters may contain several UML diagrams of the same kind. Therefore, we can use the same approach for cluster development and for collaborative development of several clusters. Contracts can directly be transformed to corresponding rules of the cluster management ASM^{Cluster} .

For instance, the condition $c \in \zeta_{C_1, C_2}^{\text{Contract}} \text{CoII}$
 An agent is permitted to modify a cluster as long as the agent changes only those shared parts in a UML diagram cluster that do not change the input-output-behavior of any associated $\text{ASM}^{\text{Prgrm}}$.

is supported by the following rule for activities of agents a to cluster C_i :

```

PERMISSIONFORMODIFICATION ( $a, C_{i,old}, C_{i,new}$ )
if  $\text{typeOfEvent}(\text{event}(a)) == \text{Modification} \wedge$ 
     $\text{owner}(C_{i,old}) == a \wedge \text{owner}(C_{i,new}) == a$ 
  then // Activation
    if  $\text{stateOfEvent}(\text{event}(a)) == \text{tryToCommitModification} \wedge$ 
         $\text{inputState}(\text{ASM}(C_{i,old})) == \text{inputState}(\text{ASM}(C_{i,new})) \wedge$ 
         $\text{outputState}(\text{ASM}(C_{i,old})) == \text{outputState}(\text{ASM}(C_{i,new}))$ 
      then  $\text{stateOfEvent}(\text{event}(a)) := \text{permitted}$  // Execution
      if  $\text{stateOfEvent}(\text{event}(a)) == \text{permitted}$  // Finalisation
      then  $\text{CommitModification}(C_{i,old}, C_{i,new})$ 
           $\text{stateOfEvent}(\text{event}(a)) := \text{completed}$ 
  
```

We use basic functions $\text{stateOfEvent}(e)$, $\text{typeOfEvent}(e)$ for events, $\text{event}(a)$ for agents, $\text{owner}(C)$ for clusters, a number of derived functions such as the functions $\text{inputState}(\text{ASM})$, $\text{outputState}(\text{ASM})$, and the transition rules

$\text{CommitModification}(C_{old}, C_{new})$ and $\text{Negotiate}(a, C_{old}, C_{new})$.

The transition rule $\text{Change}(C_{old}, C_{new})$ can only be applied if the permission for this modification is given to an agent.

At the same time if a shared part is affected by the modification then a negotiation process must start.

```

OBLIGATIONSIMPOSEDBYMODIFICATION ( $a, C_{i,old}, C_{i,new}$ )
if  $\text{typeOfEvent}(\text{event}(a)) == \text{Modification} \wedge$ 
     $\text{owner}(C_{i,old}) == a \wedge \text{owner}(C_{i,new}) == a$ 
  then // Activation
    if  $\text{stateOfEvent}(\text{event}(a)) == \text{tryToCommitModification} \wedge$ 
         $\text{inputState}(\text{ASM}(C_{i,old})) == \text{inputState}(\text{ASM}(C_{i,new})) \wedge$ 
         $\text{outputState}(\text{ASM}(C_{i,old})) \neq \text{outputState}(\text{ASM}(C_{i,new}))$ 
      then  $\text{Negotiate}(a, C_{i,old}, C_{i,new})$  // Execution
           $\text{stateOfEvent}(\text{event}(a)) := \text{negotiate}$ 
      if  $\text{stateOfEvent}(\text{event}(a)) == \text{negotiate} \wedge$ 
           $\text{stateOfNegotiation}(a) = \text{completed}$ 
      then  $\text{CommitModification}(C_{i,old}, C_{i,new})$  // Finalisation
           $\text{stateOfEvent}(\text{event}(a)) := \text{completed}$ 
  
```

In a similar form we develop sanctions for the case of the violation of a contract. The most liberal sanction is the delivery of a copy to the agent that has been inconsistently modifying the cluster. The collaboration with other agents is then interrupted until the cluster copy is again associated with others by a contract.

We may generalise this approach to contract managers as discussed in Section 3.2. The rules above may be generalised to

```

PERMISSIONFORMODIFICATION ( $a, C_{i,old}, C_{i,new}, \zeta^{\text{Contract}}$ ) and
OBLIGATIONSIMPOSEDBYMODIFICATION ( $a, C_{i,old}, C_{i,new}, \zeta^{\text{Contract}}$ ).

```

4.3 Support for Change Management

The constraint $EC_1^{states(SC,CT)}$ requires that states used in a statechart must be states of the ruling class. Let us now assume that one agent decides to decompose a ruling class. For instance, it becomes known that the lending process for a student is different from that lending process of an employee. Employees can borrow a book for an unlimited period. The student must return the book at least by the the deadline. We thus are going to modify cluster C_2 . People considered are either students or employees.

Since the class *Person* is split into two classes the *statechart* diagram in C_2 is going to be modified. The new classes inherit the attributes of the old class *Log4BorrowingABook*. This change is performed according to the rules for collaboration discussed in Section 4.2 The result is given in Figure 3. Finally, the *statechart* diagram is harmonised with the *Log4BorrBookEmp* and *Log4BorrBookStud* classes. The integrated $ASM_{styleAaM}^{Prgrm}$ is obtained as a result of these operations.

This example shows now that our framework prohibits from developing inconsistent clusters.

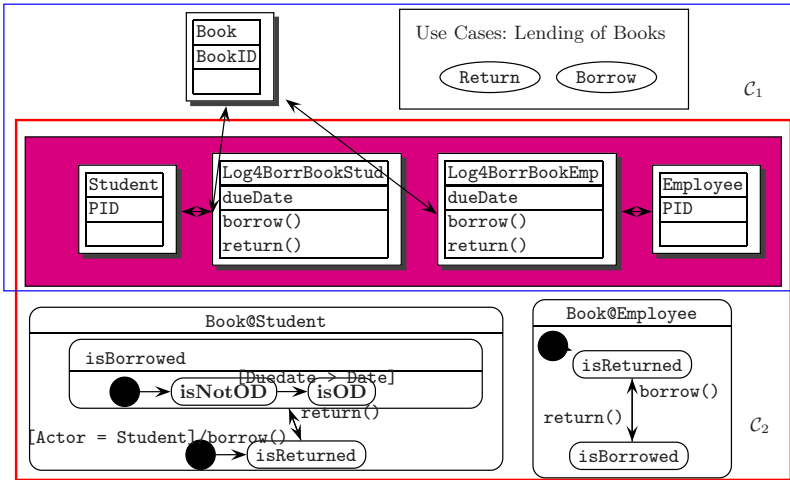


Fig. 3. Diagram clusters after modification (separation of class *BorrowingABook*)

5 Conclusion

Software is typically a complex product that has been developed by a team. The development process is often similar to the work of an artisan. Intermediate products are typically rather informal and may be contradictory. The final product is a program that has a well-defined semantics. Large software systems can be simplified tremendously if techniques of modular modelling such as design by

UML diagram clusters are used. Modular modelling is an abstraction technique based on principles of hiding and encapsulation. Design by UML diagram clusters and its corresponding \mathcal{ASM} allows to consider parts of the software systems in a separate fashion. Software reuse has been considered but never reached the maturity for application engineering.

It is important to provide a consistent and unambiguous semantics for software specification within the team context, so that all team members have the same interpretation of the specification. The fact that UML lacks a precise semantics is a serious disadvantage of UML-based methodologies.

UML might use semantic variation points [CJ05] for support of intentional degrees of freedom for interpretation of the metamodel semantics. Semantic variation points are used for a family of languages sharing commonalities and some variabilities that one can customize for a given application domain. This UML approach does, however, neither solve the consistency problem nor the problems 1 to 4.

UML suffers typically from the non-integrated development of different UML diagram types. Currently some of the diagrams such as use cases and state-chart diagrams are associated by mappings. These mappings are however not refinements [BS03] in the sense of the \mathcal{ASM} approach, for example, changes in state-chart diagrams do not result in changes of use case diagrams. Often diagrams are not associated at all, see, for instance, the lacking association between class diagrams and use case diagrams in Figure 1.

This paper contributes to consistency management of clusters of UML diagrams. Our solution has a number of advantages:

Faithful coexistence of UML diagrams: As long as we have been choosing faithful representations of the $\mathcal{ASM}^{\text{Prgm}}$ by a number of UML diagrams we may bind any change of one of the UML diagrams by $\mathcal{ASM}^{\text{Contract}}$. The main aim of these contracts is to support consistency among these diagrams.

Contract-based refinement of specifications: Whenever a refinement is applied to the specification then the refinement is only committed if all contract conditions are satisfied. Otherwise we apply an enforcement method such as cascading refinement or default refinement/modification to other parts of the specification, such as rejection of the current refinement or such as the derivation of obligations for later refinement steps.

Co-evolution of UML diagram clusters: UML development methodologies often use a number of diagrams at the same time. Their integrated evolution has not been satisfactorily solved so far. A solution cannot be envisioned due to open semantics of UML diagrams. We use \mathcal{ASM} specifications as the backing specification and demonstrate how UML diagrams can co-evolved and coexist.

The main achievement of our approach is consistency management during the development process. Developers can use the large variety of UML diagrams. Utilization of such varieties of UML diagrams is often required by industrial partners at the moment and is thus a convincing argument in favor for UML.

Acknowledgement. We thank Egon Börger and Klaus-Dieter Schewe for their comments, discussions, and advices.

References

- [BCR00a] Börger, E., Cavarra, A., Riccobene, E.: An ASM semantics for UML activity diagrams. In: AMAST, pp. 293–308 (2000)
- [BCR00b] Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: Abstract State Machines, pp. 223–241 (2000)
- [BGS⁺03] Barnett, M., Grieskamp, W., Schulte, W., Tillmann, N., Veanes, M.: Validating use-cases with the AsmL test tool. In: QSIC, pp. 238–246 (2003)
- [Boe76] Boehm, B.W.: Software engineering. IEEE Trans. Computers 25(12), 1226–1241 (1976)
- [Bör03] Börger, E.: The ASM refinement method. Formal Aspects of Computing 15, 237–257 (2003)
- [BS03] Börger, E., Stärk, R.: Abstract state machines - A method for high-level system design and analysis. Springer, Berlin (2003)
- [BST02] Bidoit, M., Sannella, D., Tarlecki, A.: Architectural specifications in CASL. Formal Asp. Comput. 13(3-5), 252–273 (2002)
- [CJ05] Chauvel, F., Jézéquel, J.-M.: Code generation from UML models with semantic variation points. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 54–68. Springer, Heidelberg (2005)
- [DT01] Düsterhöft, A., Thalheim, B.: Conceptual modeling of internet sites. In: Kunii, H.S., Jajodia, S., Sølvberg, A. (eds.) ER 2001. LNCS, vol. 2224, pp. 179–192. Springer, Heidelberg (2001)
- [EB04] Elaasar, M., Briand, L.: An overview on UML consistency management. Technical Report SCE-04-018, Ottawa University (2004)
- [FRT05] Fiedler, G., Raak, T., Thalheim, B.: Database collaboration instead of integration. In: APCCM 2005 (2005)
- [HKKR05] Hitz, M., Kappel, G., Kapsammer, E., Retschitzegger, W.: UML @ Work, 2nd edn. dpunkt, Heidelberg (2005)
- [ISO01] ISO/IEC. 9126-1 (Software engineering - product quality - part 1: Quality model). ISO/IEC JTC1/SC7 N2519 (2001)
- [KN02] Kollingbaum, M., Norman, T.: Supervised interaction - create a web of trust for contracting agents in electronic environments. In: Proc. of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems, pp. 272–279. ACM Press, New York (2002)
- [Mal70] Malzew, A.I.: Algebraic systems. Nauka, Moscow (1970)
- [Obe01] Ober, I.: An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions. PhD thesis, Polytechnique de Toulouse (2001)
- [pUM07] The precise UML group (2007), <http://www.cs.york.ac.uk/puml/>
- [Rei84] Reichel, H.: Structural induction on partial algebras. Mathematical research, vol. 18. Akademie-Verlag, Berlin (1984)
- [SMSJ03] Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
- [ST93] Schewe, K.-D., Thalheim, B.: Fundamental concepts of object oriented databases. Acta Cybernetica 11(4), 49–81 (1993)

- [ST99] Schewe, K.-D., Thalheim, B.: Towards a theory of consistency enforcement. *Acta informatica* 36, 97–141 (1999)
- [ST07] Schewe, K.-D., Thalheim, B.: Development of collaboration frameworks for web information systems. In: *IJCAI 2007 (20th Int. Joint Conf on Artificial Intelligence, Section EMC 2007 (Evolutionary models of collaboration), Hyderabad*, pp. 27–32 (2007)
- [Tha00] Thalheim, B.: *Entity-relationship modeling – Foundations of database technology*. Springer, Berlin (2000)
- [Tsi00] Tsiolakis, A.: Consistency analysis of UML class and sequence diagrams using attributed graph grammars. Technical Report 2000/3, Technical University of Berlin, Computer Science (2000)
- [Ves05] Vestenicky, V.: Schema integration as view cooperation. PhD thesis, Charles University Prague, Computer Science (2005)