Model Engineering

Recent Papers

Bernhard Thalheim
Christian-Albrechts-University Kiel, Department of Computer Science, 24098 Kiel, Germany
thalheim@is.informatik.uni-kiel.de

Model Engineering:

Model Suites

Typical system specification consists of a number of models for different facets and aspects of the system. UML-based development uses for instance at least half-dozen different diagrams which coherence is not given in an explicit form. Sometimes, systems description is based on a variety of abstraction levels. These abstraction level refer to each other in some fuzzy informal way.

This paper aims in the development of a theory of multi-model development. We introduce model suites as a set of models with explicit associations among the models, with explicit controllers for maintenance of coherence of the models, with application schemata for their explicit maintenance and evolution, and tracers for establishment of their coherence. This theory has been tested against typical applications such as OLTP-OLAP architectures, multi-model suites at the same abstraction layer, and challenging applications for scientific databases.

Model suites are based on a general theory, on a specification technology and on an implementation technology for sets of models that share common submodels, that collaborate and that evolve over time, with new or corrected data and with new analysis tasks. These models must be tightly coupled. Model suites are used to specify this model coupling and model collaboration. Renewable and evolving model clusters are going to be developed for the development of a theory and supporting technology.

Collaboration and Distribution of Models

Specification of distribution has neglected over a long period. Instead of explicit specification of distribution, multi-database systems and federated database systems have been extensively discussed in the literature. From the other side, database research has succeeded in developing approaches that incorporate conceptual specification and allow to reason on systems at a far higher abstraction level. With the advent of web information systems systems became naturally
distributed. Therefore, we need techniques for conceptual description of distribution. Distribution does not stand alone but follows computations and business needs. Thus, we need to consider structuring, functionality and distribution at the same time. Since these aspects are intertwined with each other and systems cooperate, communicate and coordinate their action we base our consideration on collaboration. It integrates communication, coordination and cooperation. In this paper we develop a specification framework for collaborating systems.

Table of contents

1. Model Suites
   published in [Tha08b]
   related papers: [Tha08a,Tha07c,JTK+09,Tha07b,MT07,LT06,MSTZ04,MKDTZ05]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

2. Co-Evolution of (Information) System Models
   published in [DT10a]
   related papers: [DT10b,ST10,Tha10b]

3. Towards ASM Engineering and Modelling
   published in [ST07b]
   related papers: [Tha07a,BT08,ST08a,RNS+07]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

4. ASM-Supported Management of UML Clusters
   published in [ST09]
   related papers: [MT07,MST05,FT02,ST04b]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

5. Engineering Database Component Ware
   published in [Tha07b]
   related papers: [Tha07a,LT05]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

6. Development of Collaboration Frameworks for Distributed Web Information Systems
   published in [ST07a]
   related papers: [Tha07a,Tha08a,ST06,ST04a,ST05a]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

7. The Enhanced Entity-Relationship Model
   published in [Tha09b]
   related papers: [Tha00,Tha07a,ST08b,ST08a,Tha07c,DMT07,MDT04]
   see also my website under miscellaneous or Verschiedenes: talks 2007/2008

8. Generalisation and specialisation
   published in [Tha09b]
related papers: [Tha00, BT07]
see also my website under miscellaneous or Verschiedenes: talks 2007/2008

9. Abstraction
published in [Tha09b]
related papers: [Tha00]
see also my website under miscellaneous or Verschiedenes: talks 2007/2008

Papers on the theory of (conceptual) models, the theory of (conceptual) modelling activities, theory of (conceptual) modelling, and applications of a theory of modelling are not in this collection. See also [Tha11, Tha10a, Tha09a] and for applications of this concept [KZK+10, ST05b, TSM09, FJM+09, MST09].

References


H. Ma, K.-D. Schewe, and B. Thalheim. Integration and cooperation of media types. In ISITA'05, 2005.


Model Suites

Bernhard Thalheim

Department of Computer Science, Christian-Albrechts-University at Kiel,
Olshausenstrasse 40, 24098 Kiel, Germany
thalheim@is.informatik.uni-kiel.de

Abstract. Typical system specification consists of a number of models for different facets and aspects of the system. UML-based development uses for instance at least half-dozen different diagrams which coherence is not given in an explicit form. Sometimes, systems description is based on a variety of abstraction levels. These abstraction level refer to each other in some fuzzy informal way.

This paper aims in the development of a theory of multi-model development. We introduce model suites as a set of models with explicit associations among the models, with explicit controllers for maintenance of coherence of the models, with application schemata for their explicit maintenance and evolution, and tracers for establishment of their coherence. This theory has been tested against typical applications such as OLTP-OLAP architectures, multi-model suites at the same abstraction layer, and challenging applications for scientific databases.

Model suites are based on a general theory, on a specification technology and on an implementation technology for sets of models that share common submodels, that collaborate and that evolve over time, with new or corrected data and with new analysis tasks. These models must be tightly coupled. Model suites are used to specify this model coupling and model collaboration. Renewable and evolving model clusters are going to be developed for the development of a theory and supporting technology.

1 From Loosely Coupled Model Sets to Synchronised Model Suites

1.1 Classical Database and Information Systems Modelling

Classical database or information systems modelling is based on some modelling language. Additionally and often in a hidden form, restrictions are added to the modelling language. The model of the reality is then developed in a negotiation and scoping process. The result is a model with undocumented decisions about scoping, restrictions, theories behind, reference models that have been used intentionally, and about rigidity of the notions used in the model. Some modelling approaches use a systematic way for building models and reach a higher level of maturity [JMTV05].

Formal languages are typically defined through their syntax and semantics [ST08a] by an inductive definition of well-formed expressions over an alphabet in certain language. Pragmatics is often assumed but not provided in an explicit form. We also observe that models use a variety of semantics such as lexical semantics for naming of model constructs, grammatical semantics for associations among constructs, and logical semantics for integrity constraints. This mixture of semantics results in a variety of logics, e.g., deontic, epistemic, modal, belief, and preference logics. Furthermore shortcuts, ambiguities, and ellipses are used for making...
models more compact and at the same time context-dependent. Models use also their inherent language semantics and constraints. At the same time, information system theory uses classes of constraints instead of sets of real-life constraint. The modelling process is a negotiation process, uses identification and analysis of the barriers at various abstraction layers such as strategic, psychological, legal, and structural layers and tries to find an acceptable solution to resolution of conflict.

Modelling of information systems has been considered for a long time only at the structural level and neglected the impact of behaviour to modelling decisions. Database theory requires normalisation of structures and database practice often uses denormalisation. Nowadays, it is accepted [Tha04] that modelling must be based on an integrated process that considers structuring and functionality as different sides of the same coin.

Modelling is an incremental process. Therefore, a number of mapping and refinement notions for varying granularity of concepts have been developed. Typically, information systems modelling can be started with a specification of structuring of the information system. Generic functionality can be added for formulation of functions, views, queries, and maintenance as long as these functions are atomic. Complex functions need their specific specification. Finally the limitations, specific opportunities and special functionality of implementation platforms is taken into account for realisation of the system.

The classical database specification process uses a number of languages depending on the abstraction layer, on the aspect under consideration, on the implementation platform and on the communication language. This variety is surveyable as long as simple applications are targeted. Whenever functionality becomes more complex workflow specification languages such as BPMN [BT08a, BT08b] are used. Whenever interfaces are under consideration, web or interface languages are used. Whenever collaboration of systems is used, distributed systems or clusters of systems are developed at the the implementation level.

1.2 Living in Models

Manyfold of Coexisting Models.

The co-design of information systems is based on a model for structuring of the system, another model for functionality, a third one for distribution, and a fourth one for interaction. The co-design approach [Tha00] is the first approach that supports an integrated model development. The basis of this integration is a homogeneous model language. Information systems development is however often still based on inhomogeneous models that concentrate on one of the aspects.

This variety of models results in a mixture of models. At the same time, various abstraction levels must be considered. For instance, scientific data are often based on sensor or streaming data. These data are used as micro-data for the generation of (meso-)data that are going to be stored in the database. Aggregated data are used for specific analyses which again are used for the derivation of data such as OLAP data. Finally, these OLAP data are used for computation of display (macro-)data.

A similar observation can be made for UML based modelling. System specification may start with use case diagrams, be based on class, component, interaction, statechart, etc. diagrams and may be mapped to specific implementation languages. Typically, diagrams remain on their specific level of detail and are not updated whenever another diagram is updated. Their association can however be based on a local as view approach [ST08b] that uses an
abstract state machine for the system specification and diagrams as specific views for consideration of specific aspects.

A Reference Example of Multi-Scale and Multi-Abstraction Modelling.

Model suites for heart modelling consists of a 5-layer model of the heart [HLMN06]. At the genes layer the networks of genes are given by molecular functions. Proteins form the elementary units, define the chemistry, and their composition. Cell structure are the basis elements for explanation of functions and key organisational unit with biological processes and pathway models. The tissue model describes the structure and function and with cellular components. The body is described by a system of myocardian activation. The time range varies by $10^{15}$ and the space range by $10^9$.

The 5-layer model of the human heart is based on different interacting “suites”. These model suites are built up by levels for genes (networks based on molecular function), proteins as elementary units and their composition, cells (focal level, pathway model), tissues and myocardial activation.

This multi-abstraction layer model set is based a very different but nowadays integrated models. The biophysics model of nerves and muscle is based on cable theory, ionic currents, Hodgkin-Huxley equations, muscle models, and fading memory model. The cardiac electrophysiology model uses cardiac cells, units, the diFrancesco-Noble model, membrane models, and bidomain models. The electrocardiography model is used to describe cardiac anatomy and function, activation, body surface potential mappings, transfer matrices, myocardial inverse procedures, and normal and abnormal ECG.

1.3 From Many-Model Modelling to Model Suites

This paper aims in the development of a theory, specification framework and of an implementation approach for coherent management of a set of models. Sets of models are coherent if they are logically consistent or integrated, if they have the quality of cohering, especially are coordinated by a coherent plan for usage and evolution, and if their association is explicit and maintained.

A model suite consists of a set of data models, an integration or association schema and obligations requiring maintenance of the association among the models. Model suites are an extension of model ensembles used for distributed or collaborating databases. Ensemble databases form a group of databases that support a single effect in an application. Ensemble databases are based on an homogeneous platform and often have a common database modelling language. Model suites also generalise model clusters which are mainly a group of models forming a unit or constituting a collection. From the other side, model suites are an extension of model farms [YTS99, STY99] that are oriented towards collaboration with check-in and check-out facilities and corresponding flexible networking. The model suite we envision is based on historical and spatial data and displays a number of viewpoints or interpretations among these data. Each of the data models is going to be characterised by a number of quality criteria.

The paper aims to develop languages for model suites. A basis for such languages might be the extended entity-relationship model [Tha00]. Entity types of this models may accommodate suites of indicators or properties. Relationship types may accommodate associations
among these indicators. Suites and associations are however a matter of change, evolution and revision. Therefore, we use a model suite for the model of dependencies among indicators.

1.4 Outline of the Paper

The model suite for heart modelling is not at all an exceptional application case. Section 2 discusses other application cases where model suites could lead to a consistent and manageable solution for modelling problems. The notion of model suites is introduced in Section 3. It generalises the notion of clusters [ST08b]. Section 4 discusses variations and specific model suites. We concentrate in this paper on the conceptual description of model suites and defer illustrating examples to another paper.

2 Challenging Applications

2.1 Ecological Systems

Ecological systems are very complex systems. These systems are characterised by a large number of properties that can only be partially observed and recorded. Their association is currently almost unknown. These systems depend on natural systems, on political systems, bodies of knowledge that have been used at different historical periods, and common skills on technology within a historical period and observed for a certain region.

Modelling of ecological systems is currently based on indicators. These indicators are partially correlated, or refine other indicators depending on the level of detail currently considered, and constitute basic units of ecological models. They thus form elementary components of models. These models must be tightly coupled. Their coupling, correlation and interdependence is a primary research goal. We may distinguish these models depending on the abstraction level such as general society or climate models at the most general level of detail, regional models as more specific models, models of ecological fields as models in the small, models of species and models of chemical exchange at the most specific level of details.

These models form a suite of coupled and interdependent models, i.e. a suite of interacting models. Their interrelation is based on an explicit specification of their collaboration. This collaboration may be represented based on their aspects for their evolution and adaptation: communication of evolution, coordination of evolution and cooperation during evolution. The communication supports to exchange values among the models. Coordination regulates this exchange. Cooperation models the co-evolution of some models in the suite. Model suites are used to specify this model coupling and model collaboration.

The collaboration of models is based on

(1) mappings for abstracting from more specific models to more general ones, for refining and specialising more general models to more specific models, and for associating models at the same level of detail,

(2) constructors for building models based on indicators, for explicit treatment of their correlation and disassociation, and

(3) contracts among models that allow to simulate co-evolution of models.
Mappings to be developed must support also association of analog or continuous models to discrete ones. These mapping allow to summarise, to filter, to scope and to abstract microdata of models at the lowest level of detail to mesodata models. They must be insensitive against injection of data that are specific at the target level but not considered at the source level. Typical constructors for models are shuffle product, reduct, scope and integration operators. Contracts are the main element for association and integration of models at various levels of detail.

Ecosystem challenge current modelling techniques by

- a large variety of changes in scope, in impact, in granularity, in determinacy, in abstraction level, in data, etc.,
- by requiring modelling support for continuous evolution of systems and of models, and
- by the necessity to handle model interactions at all levels.

2.2 Environmental Sciences

The Kiel Research Cluster of Excellence ‘Future Ocean’ uses a large model suite for a large variety of problems to be handled. The Cluster investigates the ocean change and its consequences, ocean resources and opportunities, and the ocean risks and management. It aims in reconstructing the past, to investigate ongoing changes in the ocean, and to predict future changes depending on scenarios assumed for evolution. For instance, the mankind’s impact on the ocean is substantial. Increasing temperatures as a result of a global climate change are leading to a sea level rise and change of living conditions for marine organisms. The increasing uptake of carbon dioxide and other greenhouse gases is acidifying the ocean and leads to long-term effects on biological and chemical processes. Some fish stock is over-fished, species are going extinct and the ecological balance in the ocean is shifting. The coasts are becoming more heavily populated and exploited. The results in an increase in pollution threatens coastal zones.

The ocean offers a huge diversity of mineral and biological resources. Sustainable use of these natural resources will benefit biodiversity and mankind. At the same time, floods, tsunamis, sea level rise or submarine earthquakes are increasingly threatening the growing population in coastal zones.

The Research Cluster uses biological, chemical, biological, geological, geographical, physical, mathematical etc. models. Some of the models are well-understood. Most of the models are exploratory models and are currently under constant revision and change. These models are interdependent on each other. Some of the models use a very high-level of abstraction, other very low level of abstraction and very fine-grained data.

The Research Cluster uses very large data sets for each of these models. The data sets are of varying quality and are sometimes very dense and mostly very sparse. Most models have their own data which are not integrated with other data of other models.

Typical model suite constituents in the Research Cluster are

- mechanics models, e.g., finite element methods, integral formulation, Galerkin method, Gaussian quad,
- evolution formulas based on the theory of evolution,
- kinematics models based on transformations,
- equilibrium equations,
• constitutive equations, and
• factors models for modelling of stress, etc. etc.

2.3 Data Problems for Model Suites

Models in these applications are typically based on sparse data sets. Some of these model have also super-large data sets. Data are a very valuable resource but often of low quality. Therefore model suite handling must also supported by sophisticated concept of data management.

Data management must support controlled import of biological, chemical, geological,... data with support of common data formats, with integrity checks, with annotation of parameters, with description of experiments and surveys, and semantical checks, e.g. for data plausibility. Data of different models in the model suite must be associated with each other. At the same time various levels of data abstraction (e.g., experiment versus surveys), geographically related data sets, annotation of pictures and videos, linking publications with raw and aggregated data must be supported. There is no common approach to failed experiments and their evaluations.

Handling of data sets in a model suites allows to detect who is working on a similar topic, whether surveys are available for hypotheses under consideration and whether anybody has already rejected the current hypothesis. Controlled output of biological, chemical, geological,... data must be based on a standardisation to common formats and common usage of the data with trackable reuse. Model suites also need data visualization for producing reports in a form of geographically browseable data (e.g. GoogleEarth) and also of diagrams and plots.

Quality properties of data of models have been neglected for a long time. At the same time, modern data management allows to handle these problems. The critical findings of [PPJ06] can also be observed for data analysis and mining in environmental sciences. Therefore we have to shuffle an explicit error model [YTS+99, STY99] into the model suite thus making the error treatment and assumptions explicit. Model suite management enables us to overcome these problems and to develop a platform for high quality analysis and mining in environmental databases.

2.4 Requirements for Model Suite Handling

Model suites thus must allow to solve the following problems:

Requirement 1. Explicit specification of model suite collaboration: Interdependencies among models must be given in an explicit form. The consistency of models must be recursive.

Requirement 2. Integrated development of different models: Models are used to specify different views of the same problem or application. They must be used consistently in an integrated form. Their integration must be made explicit. Simultaneous updates of models must be allowed.

Requirement 3. Co-evolution of models: The model suite must allow data exchange between models. Changes within one model must be propagated to all dependent models.

Requirement 4. Combining different representations with mathematical rigor of models: Each model must have a well-defined semantics as well as a number of representations for display of model content. The representation and the model must be tightly coupled.

Requirement 5. Evolution of different representations: Changes within any model, must either be refinements of previous models or explicit revisions of such models. These changes must be enforced
Requirement 6. Management of model suites: The propagation of changes must be supported by scheduling mechanisms, e.g., ordering of propagation of model changes. The management must support rollback to earlier versions of the model suite. The management should also allow model change during propagation.

Requirement 7. Version handling for model suites: Model suites may have different versions.

3 Model Suites

Model suites can be defined on the basis a abstraction tower model. We distinguish between the instance layer that is populated by objects, the schema or model layer that provides a description of the coding and functionality of the objects, the language layer that defines the way how expressions and build and provides at least a syntax and semantics, and finally the theory layer for reasoning of the language. An example of such layering is the tower of object-relational data, the ER schema, the ER model language, and set theory. Another example are things, topic landscapes, the language of topic maps, and the natural language.

A model suite consists

- of set of models \( \{M_1, \ldots, M_n\} \),
- an association or collaboration schema among the models,
- of controllers that maintain consistency or coherence of the model suite,
- of application schemata for explicit maintenance and evolution of the model suite, and
- of tracers for the establishment of the coherence.

Coherence describes a fixed relationship between the models in a model suite. Two models are coherent when each change in one of the models is propagated to the other model. This change transfer implicitly assumes that the integrity constraints of the corresponding model types remain to be valid. They are non-coherent if there is a random or changing relationship. We aim in an explicit specification of the association schema and use an explicit specification of the collaboration among models. For instance, the master-slave association or collaboration propagates any change of the master to its slaves. Slaves do not have any right the change the master without consensus with the master.

We observe that this notion of model suites combines different abstractions. For a systematic treatment and introduction we shall separate the description level, the control level, the application level and the observer level.

We use an abstraction layer model for the definition of model suites. We assume that models can be defined in an inductive form. In this case we can distinguish between the model language layer that defines the form how models of certain language can be built, the model algebra layer that allows to define the set of all models under consideration and operations for changing the model, model suite layer that allows to reason on the models currently under consideration, and the model objects layer that consists of the instances of the model. Typically, the language layer starts with a signature for the definition of the language. This signature may already use some kind of lexical semantics [ST08a] for the name space used for denoting elements of the alphabet.
Models are defined within a language. The explicit collaboration of models is based on constructors, mappings, and contracts among the models.

Models can describe a variety of aspects. For instance, the entity-relationship model concentrates on the description of structures, has a theoretical, formal and graphical form for specification and is used for prescription of structuring of an object-relational database. Other models concentrate on functionality of behaviour. Therefore, we distinguish between the aim of the model, the aspect to be considered and the abstraction level for treatment and reasoning on models. These three dimensions of models can be combined within a model in a form depicted in Figure 1.

![Figure 1: Dimensions combined within a singleton model](image)

### 3.1 Model Languages

The notion of model suites generalises the notion of model clusters [ST08b] that has been used for coherence management of UML diagram sets. It combined with the theory of institutions and the CASL approach [BST02].

Typically a model is defined in a certain language. A model language $\mathcal{L}$ for a model uses some signature $\mathcal{S}$ and a set of constructors $\mathcal{C}$ that allows to build a set of all possible expressions in this language. Typically constructors are defined by structural recursion [Tha00]. The set of constructors may allow to build expressions that do not fulfill certain quality or more generally integrity conditions. Therefore we introduce a set $\Sigma_{\mathcal{S},\mathcal{C}}$ well-formedness conditions.

Languages allow the description of (complex) words or expression in the language. This language can then be extended to a logics of this language. We may use any logical language. This paper restricts languages to the first-order predicate calculus. Languages of continuous models are therefore out of the scope of this paper.

A model type $\mathcal{T}_{\mathcal{S}} = (\mathcal{L}_{\mathcal{S}}, \Sigma_{\mathcal{L}_{\mathcal{S}}})$ is defined by a pair consisting of the language of the model and by constraints $\Sigma_{\mathcal{L}_{\mathcal{S}}} \in \mathcal{L}(\Sigma_{\text{WellFormed}})$ applicable to all models defined in the given language.

Model languages $\mathcal{L}_{\mathcal{S}_1}, \ldots, \mathcal{L}_{\mathcal{S}_n}$ may be bound to each other by partial mappings $\mathbb{R}_{i,j}$:
based on their signatures. These mapping typically define the association of elements among the languages.

Model languages may be tightly coupled with each other. For instance, the representation language of ER types uses rectangles and diamond for entity types and relationship types.

A model is based on an expression in the given language. Typically, it has a structure definition, a semantics definition, and a pragmatics definition. Semantics restricts the models we are interested in. Pragmatics restricts the scope of the users of models. We explicitly define a model $M$ by an expression $\text{struct}_M$ in a language $L_S$ that obeys $\Sigma_{L_S}$, by a set of constraints $\Sigma_M$ defined in the logics of this language. Therefore, each model has its model type. We denote by $M_T$ or $M_i$ for some $i$ the set of all models of this type.

### 3.2 The Model Construction and Association Algebra

We already postulated that a model suite has models and an association schema that declares the association among these models. This association schema must be constructive in the sense that the mappings $R_{i,j}$ allow to define which component of which model is mapped to which component of another model.

Additionally models may be defined through combination of (less complex) models. Therefore, we need an algebra for construction of models. Finally, models may be associated with each other. Therefore, we define abstract predicates for the association of the models.

The model construction and association algebra $(\mathcal{M}, \mathcal{O}, \mathcal{P})$ consists of

- a manyfold of models $\mathcal{M}$ of model types $T_{L_{S_1}}, \ldots T_{L_{S_m}}$ under consideration,
- algebraic operations $\mathcal{O}$ for computing complex models such as combination $\boxdot$ of models, abstraction $\Box$ of models by projections, quotient $\Box$ of models, renaming $\rho$ of models, union $\cup$ of models, intersection $\cap$ of models, minimal negation $\rightarrow$ of a model within a given context, and
- predicates $\mathcal{P}$ stating associations among models such as the sub-model relation $\preceq$, a statement $\exists$ whether models can be potentially associated with each other, a statement $\exists$ whether models cannot be potentially associated with each other, a statement $\exists$ whether models are potentially compatible with each other, and a statement $\exists$ whether models are incompatible with each other.

The combination of two models results in a model that has all components of the two models. The abstraction is used for a reduction of components of a model. The quotient allows to concentrate on those components of the first model that do not appear in the second model. The union takes all components of the two models and does not combine common components into one component. The full negation allows to generate all those components that do not appear in the model. The minimal negation restricts this negation to some given context.

We require that the sub-model relation is not transitively reflexive. The compatibility and incompatibility predicate are not contradicting. The potential association and its negation must not conflict.

The predicates should not span all possible associations among the models but only those that are meaningful in a given application area. We may assume that two models are either potentially associated or cannot be associated with each other. The same restriction can be made for compatibility.
This model world is very general and allows to derive more advanced operations and predicates. If we assume completeness of compatibility and association predicates we may use expressions defined by the operations and derived predicates:

The predicate $\exists a \land \exists b$ is used for diverging models.

The predicate $\exists a \land \exists b$ is used for isolated from each other models.

The predicate $\exists a \land \exists b \land \not\geq \land \not\leq$, and is used for homogenisable models.

The predicate $\exists a \land \not\exists b$ is used for heterogeneous models.

Models are either atomic or constructed models. We use a number of representations for visualisation of the models within the application domain:

*Model maps* are graphs consisting of nodes representing models and edges representing the sub-model relation and the combination of models to complex models.

*Model hierarchies* are forests representing models and their sub-model relation.

*Full model maps* are graphs consisting of nodes representing models and edges representing the sub-model relation and the combination, abstraction, union, intersection, quotient, full negation and minimum negation of models if these models are used within the application.

The extraction of application relevant models from models is supported by five operations that are easing the task of efficiently acquiring relevant requirement prescriptions and of documenting them:

1. Model *projection* narrows or scopes the model to those parts (entities or concepts, axioms or invariants relating entities, functions, events, and behaviours) that are of concern for the application relevant models.
2. Model *instantiation* lifts the general models to those that are of interest within the solution and instantiates variables by values which are fixed for the given system.
3. Model *determination* is used for selecting those traces or solutions to the problem under inspection that are the most perspective or best fitting for the system envisioned. The determination typically results in a small number of scenarios for the models that are going to be supported.
4. Model *extension* is used for adding those facets that are not given by the model but are given by the environment or by the platforms which might be chosen or that might be used for simplification or support of the model (e.g., additional data, auxiliary functionality).
5. Model are often associated, adjacent, interact or fit with each other. Model *join* is used to combine models into more complex and combined models that describe a complex solution.
We are not interested in any kind of associations. We want to propagate changes in one model to other models. These associated changes can be explicitly modelled by a collaboration contract among models.

We explicitly associate the model suite by collaboration of models. According to [SYea03], collaboration means to work jointly with others or together especially in an intellectual endeavor and to cooperate with an agency or instrumentality with which one is not immediately connected. This general understanding of collaboration can also be applied to suites of collaborating models.

We distinguish three facets of collaboration: communication, coordination and cooperation. Communication is used in a variety of facets as an act or instance of transmitting or a process by which information is exchanged between models through a common system. Coordination expresses the act or action of coordinating the harmonious functioning of models for effective results. Cooperation expresses the action of cooperating. This understanding has directly led to the 3K model of collaboration that is the basis of our understanding of collaboration.

We use the approach [ST07] developed for web information systems for explicit description of collaboration within a model suite. This model specifies three perspectives of collaboration displayed in Figure 2:

Communication is defined via exchange of messages and information or classically defined via services and protocols [Kön03]. It depends on the choice of media, transmission modes, meta-information, conversation structure and paths, and on the restriction policy.

Coordination is specified via management of individuals, their activities and resources. It is the dominating perspective of collaboration. The specification is based on the pre-/post-articulation of tasks and on the description management of tasks, objects, and time. Coordination may be based on loosely or tightly integrated activities, may be enabled, forced, or blocked.

Cooperation is the production taking place on a shared space. It can be considered as the workflow perspective.

We use these ingredients of the perspectives for the specification of collaboration.

A number of models have already been proposed for CSCW systems such as coordination theory [MC94], activity theory [KKB95], task management approaches [KHW99], action/interaction theory [FTKW95], and object-oriented conceptual models [Tee96]. We generalize these approaches and propose a more general model.
The collaboration style of a model suite is based on four components describing supporting programs of the information system including session management, user management, and payment or billing systems;

data access pattern for data release and locking through the net, e.g., broadcast or P2P, for sharing of resources either based on transaction, consensus, and recovery models or based on replication with fault management, and for remote access including scheduling of access;

the style of collaboration on the basis of peer-to-peer models or component models or push-event models which restrict possible communication;

and the coordination workflows describing the interplay among parties, discourse types, name space mappings, and rules for collaboration.

Collaboration pattern generalize protocols and their specification [Kön03]. We know a number of collaboration pattern supporting access and configuration (wrapper facade, component configuration, interceptor, extension interface), event processing (reactor, proactor, asynchronous completion token, accept connector), synchronization (scoped locking, strategized locking, thread-safe interface, double-checked locking optimization) and parallel execution (active object, monitor object, half-sync/half-async, leader/followers, thread-specific storage):

Proxy collaboration uses partial system copies (remote proxy, protection proxy, cache proxy, synchronization proxy, etc.).

Broker collaboration supports coordination of communication either directly, through message passing, based on trading paradigms, by adapter-broker systems, or callback-broker systems.

Master/slave collaboration uses tight replication in various application scenarios (fault tolerance, parallel execution, precision improvement; as processes, threads; with(out) coordination).

Client/dispatcher collaboration is based on name spaces and mappings.

Publisher/subscriber collaboration is also known as the observer-dependents paradigm. It may use active subscribers or passive ones. Subscribes have their subscription profile.

Model/view/controller collaboration is similar to the three-layer architecture of database systems. Views and controllers define the interfaces.

3.4 Contracts Specifying Coordination Within a Model Suite

Coherence of model suites 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. We base our approach on a four-layer treatment in contracted development:

1. Declaration of constraints that are applied to a singleton model or to sets of collaborating models;

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 model suites;
3. Description of change and evolution steps that can be applied for refinement or modification of sets of model suites 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 model suites within development teams. In this case, team members are supported by approaches to collaboration, e.g. explicit services and exchange frames [ST07].

A contract $\mathcal{C}$ consists of a declaration of constraints, of a description of the enforcement mechanism and of a prescription of modification steps that transform a coherent model suite into a coherent model suite.

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

- contracts declare correctness of a model suite, separate exceptional states from normal states for these model suites, and forbid meaningless model suites,
- contracts enable the direct manipulation of the model suite 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 coherence of a model suite 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 model suites.

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 models 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.

### 3.5 Model Suites

Model suites consist of models that are associated with each other. This association must be made explicit. We first define the model suite type and then define model suite.

A model suite type $ST = (T_{L_{S_1}}, \ldots, T_{L_{S_n}}, \Sigma_{L_{S_1}}, \ldots, \Sigma_{L_{S_n}})$ is given by model types $T_{L_{S_i}}$ defined on a set $L_{S_1}, \ldots, L_{S_n}$ of languages and a set $\Sigma_{S_1}, \ldots, \Sigma_{S_n}$ of constraints on these languages.

A model suite $\mathcal{S}$ on a model suite type $ST$ consists of models $(M_1, \ldots, M_n)$ of type $T_{L_{S_i}}$ that obey $\Sigma_{L_{S_1}}, \ldots, \Sigma_{L_{S_n}}$.

The contract on $\mathcal{C}$ thus consists of the constraints $\Sigma_{L_{S_1,}} \cup \ldots \cup \Sigma_{L_{S_n}} \cup \Sigma_{L_{S_1}}, \ldots, \Sigma_{L_{S_n}}$, a description of the enforcement mechanisms for any operation that can be used for modification of one model, and a set of consistent evolution transformations.
3.6 Model Change Propagation in a Model Suite

Evolution and change steps of a model suite 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 and change steps is typically a model suite.

Synchronisation of models is a difficult matter for which a general solution is unlikely to exist. [Dis08] introduces a category-based framework for direct synchronisation based on a complete embedding of submodels and context-freeness of these submodels from the other models, i.e. both models \( M_1, M_2 \) to be synchronised consist of converging submodels, i.e., \( M_1 = M_{1,0} \sqcup M_{1,2} \) and \( M_2 = M_{2,0} \sqcup M_{1,2} \) with \( M_{1,0} \sqsubseteq M_{1,2} \) and \( M_{2,0} \sqsubseteq M_{1,2} \). Moreover this model is limited by the assumption that the submodel \( M_{1,2} \) is common for both models.

Instead we generalise the database approach. Let us assume that \( M_i = M_{i,0} \sqcup M_{i,1} \) from \( \mathcal{L}_i \) and \( M_j = M_{j,0} \sqcup M_{j,1} \) from \( \mathcal{L}_j \) for models \( M_i, M_j \) that are going to be synchronised. Given furthermore a mappings \( t_{i,j} : M_{i,1} \rightarrow M_{j,1} \) and \( \ell_{i,j} : M_{i,1} \rightarrow M_{i,1} \) for which extensions of \( \mathcal{R}_{i,j}, \mathcal{R}_{j,i} \) exist in \( \mathcal{L}_i \) and \( \mathcal{L}_j \), respectively.

\[
M_i \xrightarrow{\text{extract } e_{i,j}} M_{i,1} \xrightarrow{\text{transform } t_{i,j}} M_{j,1} \xrightarrow{\text{load } \ell_{i,j}} M_j
\]

The product \( e_{i,j} \circ t_{i,j} \circ \ell_{i,j} \) of the mappings is denoted by \( \text{put}_{i,j} \). This product is neither left-inverse to \( \text{put}_{j,i} \) nor must have a right-invers \( \text{put}_{j,i} \). This phenomenon is well known for updates of views in databases [AHV95, Heg08]. Since models must obey integrity constraints of their types we might have models for which \( \text{put}_{i,j} \) is not defined. Two models \( M_i \) and \( M_j \) are called coexisting if \( \text{put}_{i,j}(M_i) \preceq M_j \) and \( \text{put}_{j,i}(M_j) \preceq M_i \). We observe that a model \( M_i \) may have many coexisting models \( M_j \).

The mappings \( \text{put}_{i,j}, e_{i,j}, t_{i,j}, \text{and } \ell_{i,j} \) may be generally given for the set of all models defined on the model types \( \mathcal{T}_{\mathcal{L}_i} \) and \( \mathcal{T}_{\mathcal{L}_j} \). In this case the sub-model embedding must be canonical.

We observe that a model \( M_i \) may have many coexisting models \( M_j \). If we use a canonical embedding then the mappings \( \text{put}_{i,j} \) can be defined on the basis of the constant complement [AHV95, Heg08], i.e., \( h(M_j, i) = M_j \sqcup e_{j,i}(M_j) \). We may now extend the mapping \( \text{put}_{i,j} \) by the constant complement of the range model and define an integration condition by \( M_j = \text{put}_{i,j}^*(M_i, h(M_j, i)) \). If the integration condition is valid for coexisting models then we may support also changes in one model and propagate the changes to the other model.

\[
M_i \xrightarrow{\text{put}_{i,j}^*} M_j \quad M_i \xrightarrow{\text{change}_i} M_i' \xrightarrow{\text{put}_{i,j}^*} M_j' \quad M_i \xrightarrow{\text{put}_{j,i}^*} M_j \quad M_i' \xrightarrow{\text{change}_j} M_i' \xrightarrow{\text{put}_{j,i}^*} M_j'
\]

We require that the mappings \( \text{put}^* \) are well-behaved, i.e. \( \text{put}_{j,i}^*(\text{put}_{i,j}^*(M_i, h(M_j, i)), h(M_i, j)) \) is defined and \( \text{put}_{j,i}^*(\text{put}_{i,j}^*(M_i, h(M_j, i)), h(M_i, j)) = M_i \).
The coexistence of models is not sufficient for change propagation. If \( M_j \) is changed to \( M_j' \) by the change operation \( \text{change}_j \) then the change diagram should commute, i.e. this change operation has a related change operation \( \text{change}_i \) that allows to change \( M_i \) directly to \( M_i' \). The change operation is typically defined on two arguments: the original model \( M_j \) and an auxiliary model \( M_{aux} \). Model suite change is called *synchronised* for \( i, j \) and a set \( O_{change}^j \) of change operations defined on \( M_j \) if for each change operation \( o_j(M_j, M_{aux}) \) from \( O_{change}^j \) a change operation \( o_i(M_i, M_{aux}) \) in the set \( O_i \) of operations on \( M_i \) exists so that the change diagram commutes for the same auxiliary model \( M_{aux} \), i.e., \( o_i(M_i, M_{aux}) = \text{put}^*_{j,i}(M_j', h(M_i, j)) \) for \( M_j' = o_j(M_j, M_{aux}) \) and \( M_i' = o_i(M_i, M_{aux}) \). The complement should be constant. Therefore we may use \( h(M_i, j) \) instead of \( h(o_i(M_i, M_{aux}), j) \).

Change operations are used for model change and evolution. The order of changes is typically important. We call two change operations \( o_i, o_j \) *liberal* to the models \( M_i, M_j \) if \( M_i' = \text{put}^*_{j,i}(M_j', h(M_i, j)) \) and \( M_j' = \text{put}^*_{i,j}(M_i', h(M_j, i)) \) for \( M_i' = o_i(M_i, M_{aux}) \) and \( M_j' = o_j(M_j, M_{aux}) \). Liberality can be extended to confluence and Church-Rosser properties [Tha00]. Liberal change operations allow to change on model and then to apply all the changes to coherent model suites.

Contract management becomes in this case rather simple. Enforcement may directly be applied to all coexisting models. We also may restrict change operation by no action, cascade, oblige enforcements. ‘No action’ means that if a change operation cannot propagated to the other model then this change operation is rolled back. Cascade enforcement requires that the other model must be changed as well. Oblige enforcement allows to delay the change operation on the other model to a later stage.

Typically the set of change operations is defined by a change pattern. For instance, all database changes can be defined through insert, delete and update. [Tha00] defines a small set of change pattern for extended ER language schemata that allow to express any change in HERM schemata.

Finally we may also require that the undo operation is also supported. This operation is nothing else than another change operation that results in restoring the old model.

### 4 Specific Model Suites

We illustrate the model suite approach to some specific model suites. Model suites can be based on very different languages. The model suite for the human heart is based at the same time on state-description models such as chemical equations, structural models, and state-transformation models and on continuous models such as differential equations for behaviour and state changes. We discuss here only the first category.

#### 4.1 Strictly Hierarchically Layered Model Suites

Hierarchically layered model use an internal architecture of the data and their grouping and abstraction within a multi-layer model such as the model of the human heart. We assume that each layer uses its own model and language. Languages in computer science can be either based on the concept of abstract data types. In this case each language consists of structuring concepts and operations that support computation of data which are structured according to these structuring concepts.

Figure 3 illustrates the layer as a building block of a multi-layered model.
Hierarchically layered models are often used in system abstraction and system building. This layering architecture can be used for the generation of data or refinement downwards and for abstraction of data upwards. Layers may have their own additional meta-data and their specific rules that control the behaviour of transformers. The general machine for transformation is depicted in Figure 4. This machine can be very complex. We might however use the Abstract State Machine approach for the specification of such transformation machines.

4.2 Hierarchically Layered Micro-Meso-Macro Data Model Suites

Data are of different granularity and precision. We distinguish the following kinds:

- Sensor micro-data are such data that have been obtained from sources. They need to be cleaned, checked, and validated according to the dependability of the data.
- Data or micro-data are stored into a data mart, data warehouse or a database. They have already been structured and validated according to a quality model.
- Macro-data are data that have already been combined on the basis of structuring operations. These data may already forgetful, i.e. we are not able to reestablish the data that have been used for their computation.
- Aggregated macro-data are data that have been computed from the macro-data or from the micro-data using aggregation functions or statistical functions.
- Annotated aggregated macro-data are obtained by enriching the data using a number of association schemata for their characterisation, for the memorisation or for their delivery to user communities.
• Founded annotated aggregated macro-data are such data that have already been validated or verified against theories and concepts. These data can be explained by the user and can be used for derivation or illustration of conclusions.

Historical data are sensor data in the best case. Often historical data are aggregated data that have already been ‘cleaned’ according to the intentions of the data provider. At a later stage the data that have been the basis of the data currently considered become available. In this case, cleansing procedures must also allow a correction of the new data in a similar form. The new data usually have a finer granularity. We are interested in sensor data. If they are not available then cleansing and data handling and processing must be robust to later corrections.

The layering model for data is illustrated in Figure 5. Sensor data are scanned and used for the generation of micro-data. Typically sensor data are full of errors, need calibration and combination with other data from other data streams. Micro-data are often too voluminous and therefore abstracted and compiled to data that are stored in a database machine. We call such data sets meso-data. These data are typically the OLTP data that are used in data warehouse applications. OLAP applications use tools for the analysis of these data and for the corresponding visualisation.

We need therefore a functionality that allows to clean data, to combine data, to aggregated data, to annotate data, and to verify data. These functions must be rather specific in our domain since the data in the project creates a number of challenges:

1. We use basic data on the region, in dependence of time of recording or elaboration and of varying quality. These data must be thus analysed and combined in such a way that their value for analysis becomes appropriate.

2. Data are partially recorded data. Records have been developed in the past depending on the goals and intentions of chroniclers. They may display outliers or catastrophes. Other data may provide progression, trend or process data.

3. Data on hand might already be aggregated data. The aggregation mechanism and the data that have been used for aggregation may be partially unknown.

The aim is thus the development of layered data models that carry an information on the quality of data, on the annotation of data, and on the verification of these data.

We use already annotation schemes for characterisation of data types. The following characteristics are going to be extended to historical and regional data, to their aggregations and combinations:
• Ordering of data may be applied using a number of ordering schemata at the same time. New ordering schemata not applied in database management must be developed since ordering of historical data is often based on associated data such as history, region, and important events.

• Defaults are changing over time. They are becoming context-dependent. Therefore, the context must also be represented whenever defaults are going to be used.

• Representation systems vary for historical periods and for different regions. They depend on agreements of the society at that time and at that place. Additionally, data representation may have been changed by the chroniclers.

• Casting of data is necessary whenever we are interested in normalised data that can be used for comparisons. The casting schema may have to be changed in the case that the new data are replacing data currently used.

• Classification is based on classification schemata that are dependent on the habits of the region and the historical period. Classification schemata carry an important part of the semantics and must thus be maintained also in the case of data aggregation.

• Inherent hierarchies (e.g. within time, space) are typical properties of data types. These hierarchies may also be used for approximation of data.

4.3 Horizontally Layered Model Suites

Model ensembles are typically homogeneous model suites defined on one common language. Their collaboration is defined by an orchestration contract that describes the role and the play of each potential model in the suite. The behaviour of model ensembles is given by specific cooperation and communication pattern, typically described by the model choreography.

The proposed approach to multi-model specification also supports incremental evolution of database systems which is a specific form of database system evolution. Facility management systems are typical application systems for which incremental evolution could be the ultimate solution instead of confusing structuring [Kah99]. Typical for such applications is the long lifespan of some of the objects. Those objects have a long history of change.

Facility management systems use a number of phases: planning phase, construction phase, realization phase, and maintenance phase. Based on meta-structures we developed the new architecture in Figure 6 that has already been experienced in a project [Raa02] by our group. In this project auxiliary databases support provide help information, information on regulations, information on customers, information on suppliers, etc.

Incremental evolution is thus supported by meta-structuring on the basis of import forms of two kinds:

Injection forms enable to inject data into another database. The forms are supported by views and view cooperation approaches. Data injected into another database cannot be changed by the importing database system. The structuring \((S^{inject}, \Sigma_S)\) of the views of the exporting database system is entirely embedded into the structuring \((S', \Sigma_{S'})\) of the importing database system. The functionality \((\mathcal{O}^{inject}, \Sigma_{\mathcal{O}})\) of the views of the exporting database system is partially embedded into the functionality \((\mathcal{O}', \Sigma_{\mathcal{O}'})\) of the importing database system by removing all modification operations on the injected data. These data can only be used for retrieval purposes.
Insertion forms enable in insertion data from the exporting database into the importing database. These data can be modified. The structuring ($S_{\text{insert}}$, $\Sigma_S$) and the functionality ($O_{\text{insert}}$, $\Sigma_O$) of the views of the exporting database system are entirely embedded into the structuring ($S', \Sigma_S'$) and the functionality ($O', \Sigma_O'$) of the importing database system.

5 Conclusion

Multi-model specification is currently the state of the art in most sciences. The integration and collaboration of these models is still an open research issue. The motivation for development of an explicit treatment of models steams from several projects of our group: the handling of ecology models; the development of coherent models that support analysis and mining in the Cluster of Excellence ‘Future Ocean’, the development of models for history databases in the Graduate School of Excellence ‘Human Development in Landscapes’, and the development of integration facilities for the integrated car portal [KBF05].

This paper generalises the approaches we have been uses and aims in explicit development of model suites. Model suites are sets of models specified in certain languages with explicit collaboration of models within an evolution or change step.

Software system modelling and specification languages are designed to provide precise abstractions of aspects of the software. Currently models for software systems do not fit neatly within a single applications and do not capture all important requirements of such systems. Therefore, developers use several models. Their combination presents a very difficult challenge. Moreover, synchronised development of models is still not solved.

A similar situation can be observed for OLTP-OLAP architectures. Classical approaches define an OLTP model for the database and then define a specific OLAP model for the OLAP architecture. The approach of [LT05] uses a hierarchically layered model suite consisting of the OLTP model and of the OLAP model with a specific contract and coherence conditions for the appropriate definition of coherence of the model suite.

The motivation for this paper has been given by classical scientific applications. In a similar form Web 2.0 applications will be based on model suites, e.g., cloud computing, content delivery architectures, public virtual worlds, knowledge management, next generation web, and knowledge wikis.
References


Co-Evolution of (Information) System Models

Ajantha Dahanayake  
Georgia College and State University, J. Whitney Bunting School of Business  
Dept. of Information Technology and Marketing, Campus Box 12, Milledgeville 31061, GA, USA  
ajantha.dahanayake@gcsu.edu

Bernhard Thalheim  
Christian-Albrechts-University Kiel, Dept. of Computer Science, 24098 Kiel, Germany  
thalheim@is.informatik.uni-kiel.de

No Institute Given

Abstract. Information systems’ modelling is based on separation of concern, facets, and aspects. Most design methodologies adopt the master-slave principle, to handle the coherence of such model assemblies. These diagrams are typically not developed from scratch and entirely until their completion. Models evolve during development and are not independent, are interrelated, and in most applications also intertwined. Their interrelationships are often not made explicit and impose changes resulting in inconsistencies to other models due to the variety of models. Therefore, the theory of model suites is introduced as a set of models with explicit associations among the models, explicit controllers for maintenance of coherence, application schemata for their explicit maintenance and evolution, and tracers for establishment of their coherence and thus to support co-evolution of information system models. The excitability is captured by integrating model suites ans MetaCASE formalisms, exploring the (modelling) method engineering and tool generation required for multi-model development.

Key words: Model suites, multi-models, model coherence, co-evolution of models, method engineering, MetaCASE.
1 Introduction

1.1 The Evolution from Holistic Modelling to Multi-Model Development of Information Systems

Information systems development includes nowadays specification of structuring, functionality, interactivity, components, distribution, etc. One might try to use a holistic approach that incorporates all aspects, facets and concerns into one language. The approach most of developers are choosing is however based on a variety of languages and models. Multi-model specification is currently the state of the art in most sciences. The integration and collaboration of these models is still an open research issue.

Already classical database modelling has been using a three-layer architecture based on a central conceptual schema and associated external schemata and at least one associated internal schema. This architecture may be extended to a multi-tier architecture. Information systems modelling is far more complex since it aims also in representation of functionality, interactivity and distribution.

Database modelling is classically defining the database dictionary, database structuring and functionality within one singleton paradigm. This approach has led to sophisticated financial services, to enterprise information systems and other database-backed practical solution which are easy to handle, relatively simple to change and to implement and which satisfied the needs of business in the 90s.

At the same time a number of applications have been developed that used the potential within the data for analysis, for exchange and collaboration of systems, e.g. OLTP-OLAP systems, decision support systems, scientific information systems, collaborative information systems and web information systems. These applications do not use a singleton language for data storage, data computation and data delivery. Their languages use different paradigms. We therefore need a way for specification of information systems applications that provide facilities for appropriate modelling depending on the needs.

The language variety of UML models is intriguing and can only be partially handled [15, 17]. Many UML diagrams do not support so far a sound foundation and have many different semantics (for instance, UML state charts have 48 different interpretations). Therefore, we need a methodology that supports multi-model development of information systems.
1.2 Exploring Crisis Response Management Information Systems Modelling

An exploratory case study was conducted at an European harbor to assess the extent of complexities during the analysis and design of a crisis response management system (CRMIS). Depending on the scale of the disaster, crisis responses in a harbor infrastructure range from dealing with a small-scale problem, in which a few organizations might be involved, to a full-scale crisis, in which multiple organizations are required to resolve and to prevent escalation of the crisis. The (re)design of the CRMIS [3] incorporated the 'virtual team' concept requiring to provide relief-response organizations with a role related picture of the crises development in time critical manner satisfying changing information needs flexibly. Virtual teams required to be extendable when a relief-response organization is required to join relief-response activities and capable of dealing with a relief-response organization when it leaves the functioning system once its task is completed. Further, required to include the structuring of advanced technologies and available technical infrastructures in a meaningful way to realize dynamic and changing user information needs during a crisis response.

Several types of modelling tools were required to arrive at the systems’ architecture. The models interpret the required and relevant solutions of information management, network knowledge and information integration in crisis response management systems designing [9]. A myriad of models were developed: models of knowledge acquisition, knowledge selection, stakeholder analysis, network models for collaboration, coordination models for relief effects, models for knowledge management, models for process descriptions, models for network and node analysis, models for defining knowledge bases and critical knowledge ownerships, and least but not last the typical information system models for data, software, integration, networks, time, space and position, security and technology and a few more.

The diagrams were incrementally developed and models evolved during development. Models were not independent, and were interrelated and in most applications intertwined. Their interrelationships were often not made explicit. Models imposed changes in other models. Changes within one model resulted in inconsistencies to other models due to the variety of models used. Resolutions were time consuming, tedious and led to project delays. Therefore, we
introduce an approach to handle co-evolution of information systems modelling as: MetaCASE toolkits for the generation of (information) systems models for a variety of modelling languages, and model suites to assure coherence of co-existence and co-evolution of (information) system models. The application of model suites concept within the disaster management to orchestrate the coherence of model assemblies is available in [19].

1.3 Co-Existence and Co-Evolution of Models

Multi-model systems development may be based on sequenced development, i.e. at each stage only one model is changed and no other model must be changed correspondingly. This situation is rather idealistic since models reflect different aspects, facets and concern and thus form a co-picture of the entire system. Their association must thus be specified in an explicit form that allows an application to model evolution itself. Additionally we may wish to cope with different abstraction layers such as requirements or conceptual layers, with different abstraction levels such as model, meta-model or meta-meta model level. Furthermore, systems development is a process itself that produces versions, components, sketches and finalised models. The basic rules of multi-model systems development can be summarized in the following fundamental principles: sovereignty of each model, equal existence of each model, and consent about other models. These principles should result in integrated evolution of models and in consentient co-evolution of different models. It follows from the co-existence of different models at a development state that, in principle, they are all equal in status. Therefore, each of the models may evolve on its own right. If however changes in one model have an impact on other models then their associations must be maintained.

Co-existence and co-evolution of models is currently a hot and difficult research topic. The literature is very rich. Typical research problems discussed at present are versioning and evolution of systems [2, 8, 14], evolution of models themselves [5, 20], approaches to refinement of models [7, 20], and multi-model management. [15, 17].

1.4 Requirements for Multi-Model Information Systems Development

Multi-model information systems development allows to concentrate on one aspects, facet or concern. At the same time, models to be used must be coherent and must co-evolve with the development process. We thus have to meet a number of requirements, e.g. the following ones:
Problem 1. Explicit specification of model collaboration: Interdependencies among models must be given in an explicit form. The consistency of models must be recursive.

Problem 2. Integrated development of different models: Models are used to specify different views of the same problem or application. They must be used consistently in an integrated form. Their integration must be made explicit. Simultaneous updates of models must be allowed.

Problem 3. Co-evolution of models: Multi-model information systems development must allow data exchange between models and explicit change propagation.

Problem 4. Management of multi-model information systems development: The propagation of changes must be supported by scheduling mechanisms, e.g., ordering of propagation of model changes. The management must support rollback to earlier versions of the model suite. The management should also allow model change during propagation.

This list of requirements may be extended to requirements such as (5) combining different representations with mathematical rigor of models, (6) evolution of different representations, (7) version handling for multi-model information systems development, and (8) explicit refinement and abstraction treatment.

1.5 Structure of the Paper

We introduce the concept of model suites in the next section. This concept allows to integrate different models developed during information systems development and to maintain coherence among models. Since models might be based on different paradigms the association concept must be very flexible. We generalise the concept of institutions that allow to integrate signatures of languages to the concept of model suites which can be based on associations among models either on the signature or the language or the model level. Section 3 develops an approach to co-evolution of models. Typically models are refined in the development process. Associated models must reflect refinements if these have an impact on those models. Section 4 challenges the concept of model suites in an application environment and demonstrates at the same time how the concept of model suites can be integrated into existing tool environments.

2 Model Suites

2.1 The General Notion of Model Suites

Model suites are an extension of model ensembles [13] used for distributed or collaborating databases [16]. Ensemble databases form a group of databases
that support a single effect in an application. Ensemble databases are based on an homogeneous platform and often have a common database modelling language. Model suites also generalise model clusters which are mainly a group of models forming a unit or constituting a collection.

A model suite consists
- of set of models \( \{M_1, \ldots, M_n\} \),
- of an association or collaboration schema among the models,
- of controllers that maintain consistency or coherence of the model suite,
- of application schemata for explicit maintenance and evolution of the model suite, and
- of tracers for the establishment of the coherence.

Coherence describes a fixed relationship between the models in a model suite. Two models are coherent when each change in one of the models is propagated to the other model. This change transfer implicitly assumes that the integrity constraints of the corresponding model types remain to be valid. They are non-coherent if there is a random or changing relationship. We aim in an explicit specification of the association schema and use an explicit specification of the collaboration among models. For instance, the master-slave association or collaboration propagates any change of the master to its slaves. Slaves do not have any right to change the master without consensus with the master.

2.2 Language Varieties for Models

Typically, a model is defined in a certain language. A model language \( \mathcal{L} \) for a model uses some signature \( S \) and a set of constructors \( C \) that allows to build a set of all possible expressions in this language. Typically constructors are defined by structural recursion [18]. The set of constructors may allow to build expressions that do not fulfill certain quality or more generally integrity conditions. Therefore we introduce a set \( S_{\text{WellFormed}} \) well-formedness conditions.

A model type \( T_{L_S} = (L_S, S_{L_S}) \) is defined by a pair consisting of the language of the model \( L_S \) of signature \( S \) and by constraints \( S_{L_S} \in \mathcal{L}(S_{\text{WellFormed}}) \) applicable to all models defined in the given language.

Model languages \( L_{S_1}, \ldots L_{S_n} \) may be bound to each other by partial mappings \( R_{i,j} : L_{S_i} \rightarrow L_{S_j} \) based on their signatures. These mapping typically define the association of elements among the languages.

A model is based on an expression in the given language. Typically, it has a structure definition, a semantics definition, and a pragmatics definition. Semantics restricts the models we are interested in. Pragmatics restricts the scope of the users of models. We explicitly define a model \( \mathcal{M} \) by an expression \( \text{struct}_{\mathcal{M}} \)
in a language $\mathcal{L}_S$ that obeys $\Sigma_{\mathcal{L}_S}$, by a set of constraints $\Sigma_{\mathcal{M}}$ defined in the logics of this language. Therefore, each model has its model type. We denote by $\mathcal{M}_T$ or $\mathcal{M}_i$ for some $i$ the set of all models of this type.

### 2.3 Model Association and Contracting in Multi-Layered Modelling

We want to propagate changes in one model to other models. These associated changes can be explicitly modelled by a collaboration contract among models. We distinguish three facets of collaboration: communication, coordination and cooperation. Communication is used in a variety of facets as an act or instance of transmitting or a process by which information is exchanged between models through a common system. Coordination expresses the act or action of coordinating the harmonious functioning of models for effective results. Cooperation expresses the action of cooperating.

The collaboration style of a model suite is based on four components describing:

- **Supporting programs** of the information system including session management, user management, and payment or billing systems;
- **Data access pattern** for data release and locking through the net, e.g., broadcast or P2P, for sharing of resources either based on transaction, consensus, and recovery models or based on replication with fault management, and for remote access including scheduling of access;
- The **style of collaboration** on the basis of peer-to-peer models or component models or push-event models which restrict possible communication;
- and the **coordination workflows** describing the interplay among parties, discourse types, name space mappings, and rules for collaboration.

Collaboration pattern generalize protocols and their specification [12]. We know a number of collaboration pattern supporting access and configuration (wrapper facade, component configuration, interceptor, extension interface), event processing (reactor, proactor, asynchronous completion token, accept connector), synchronization (scoped locking, strategized locking, thread-safe interface, double-checked locking optimization) and parallel execution (active object, monitor object, half-sync/half-async, leader/followers, thread-specific storage):

- **Proxy collaboration** uses partial system copies (remote proxy, protection proxy, cache proxy, synchronization proxy, etc.).
- **Broker collaboration** supports coordination of communication either directly, through message passing, based on trading paradigms, by adapter-broker systems, or callback-broker systems.
**Master/slave collaboration** uses tight replication in various application scenarios (fault tolerance, parallel execution, precision improvement; as processes, threads; with(out) coordination).

**Client/dispatcher collaboration** is based on namespaces and mappings.

**Publisher/subscriber collaboration** is also known as the observer-dependents paradigm. It may use active subscribers or passive ones. Subscribes have their subscription profile.

**Model/view/controller collaboration** is similar to the three-layer architecture of database systems. Views and controllers define the interfaces.

A contract $C$ consists of a declaration of constraints, of a description of the enforcement mechanism and of a prescription of modification steps that transform a coherent model suite into a coherent model suite.

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

- contracts declare correctness of a model suite, separate exceptional states from normal states for these model suites, and forbid meaningless model suites,
- contracts enable the direct manipulation of the model suite 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 coherence of a model suite 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 model suites.

### 2.4 The Notion of the Model Suite

A model suite type $ST = (T_{L_{S_1}}, ..., T_{L_{S_n}}, \Sigma_{L_{S_1}} ..., \Sigma_{L_{S_n}})$ is given by model types $T_{L_{S_i}}$ defined on a set $L_{S_1}, ..., L_{S_n}$ of languages and a set $\Sigma_{S_1}, ..., \Sigma_{S_n}$ of constraints on these languages.

A model suite $S$ on a model suite type $ST$ consists of models $(M_1, ..., M_n)$ of type $T_{L_{S_i}}$ that obey $\Sigma_{L_{S_1}} ..., \Sigma_{L_{S_n}}$.

The contract on $C$ thus consists of the constraints $\Sigma_{L_{S_1}} \cup ... \cup \Sigma_{L_{S_n}} \cup \Sigma_{L_{S_1}} ..., \Sigma_{L_{S_n}}$, a description of the enforcement mechanisms for any operation that can be used for modification of one model, and a set of consistent evolution transformations.

We use approaches developed for control theory for handling dynamics of model suites. Dynamics of layered systems is defined by pending objects, i.e. request results in initialisation of a new pending object, request being processed, and pending object issues a new request. The new request may die after issuing
the request or may wait for the response for a certain time slot with cancellation activities or may wait for the response that the request has been accepted or that is request is processed or that the request got an answer.

3 Co-Evolution of Information Systems Models Based on Model Suites

Synchronisation of models is a difficult matter for which a general solution is unlikely to exist. Instead of the category-based framework for direct synchronisation [6] we generalise the database approach. Heterogeneous models $\mathcal{M}_1, \mathcal{M}_2$ that should be synchronised consist of converging submodels, i.e., $\mathcal{M}_1 = \mathcal{M}_{1,0} \uplus \mathcal{M}_{1,2}$ and $\mathcal{M}_2 = \mathcal{M}_{2,0} \uplus \mathcal{M}_{1,2}$ with $\mathcal{M}_{1,0} \uplus \mathcal{M}_{1,2}$ and $\mathcal{M}_{2,0} \uplus \mathcal{M}_{1,2}$\(^1\). Moreover this model is limited by the assumption that the sub-model $\mathcal{M}_{1,2}$ is common for both models. For heterogeneous models we assume that $\mathcal{M}_i = \mathcal{M}_{i,0} \boxtimes \mathcal{M}_{i,1}$ from $\Sigma_i$ and $\mathcal{M}_j = \mathcal{M}_{j,0} \boxtimes \mathcal{M}_{j,1}$ from $\Sigma_j$ for models $\mathcal{M}_i, \mathcal{M}_j$ that are going to be synchronised. Given furthermore a mappings $t_{i,j} : \mathcal{M}_{i,1} \mapsto \mathcal{M}_{j,1}$, $t_{j,i} : \mathcal{M}_{j,1} \mapsto \mathcal{M}_{i,1}$ for which extensions of $\mathbb{R}_{i,j}, \mathbb{R}_{j,i}$ exist in $\Sigma_i$ and $\Sigma_j$, respectively.

The product $e_{i,j} \circ t_{i,j} \circ l_{i,j}$ of the mappings is denoted by $\text{put}_{i,j}$. This product is neither left-inverse to $\text{put}_{j,i}$ nor must have a right-invers $\text{put}_{j,i}$. This phenomenon is well known for updates of views in databases [1, 10]. Since models must obey integrity constraints of their types we might have models for which $\text{put}_{i,j}$ is not defined. Two models $\mathcal{M}_i$ and $\mathcal{M}_j$ are called coexisting if $\text{put}_{i,j}(\mathcal{M}_i) \preceq \mathcal{M}_j$ and $\text{put}_{j,i}(\mathcal{M}_j) \preceq \mathcal{M}_i$. We observe that a model $\mathcal{M}_i$ may have many coexisting models $\mathcal{M}_j$.

The mappings $\text{put}_{i,j}, e_{i,j}, t_{i,j},$ and $l_{i,j}$ may be generally given for the set of all models defined on the model types $\mathcal{T}_{\Sigma_i}$ and $\mathcal{T}_{\Sigma_j}$. In this case the sub-model embedding must be canonical.

We observe that a model $\mathcal{M}_i$ may have many coexisting models $\mathcal{M}_j$. If we use a canonical embedding then the mappings $\text{put}_{i,j}$ can be defined on the basis of the constant complement [1, 10], i.e., $h(\mathcal{M}_j, i) = \mathcal{M}_j \boxtimes e_{j,i}(\mathcal{M}_j)$. We may now extend the mapping $\text{put}_{i,j}$ by the constant complement of the range model and define an integration condition by $\mathcal{M}_j = \text{put}_{i,j}^*(\mathcal{M}_i, h(\mathcal{M}_j, i))$. If

\(^1\) $\uplus$ denotes the generalised union of models, $\uplus$ denotes separatability or divergency of models, and $\boxtimes$ denotes the generalised join.
the integration condition is valid for coexisting models then we may support also changes in one model and propagate the changes to the other model.

\[
\begin{align*}
\mathcal{M}_i & \xrightarrow{\text{put}_{i,j}^*} \mathcal{M}_j \\
\mathcal{M}_i' & \xrightarrow{\text{change}_i} \mathcal{M}_i' \\
\mathcal{M}_j & \xrightarrow{\text{put}_{i,j}^*} \mathcal{M}_j \\
\mathcal{M}_j' & \xrightarrow{\text{change}_j} \mathcal{M}_j'
\end{align*}
\]

We require that the mappings \( \text{put}^* \) are well-behaved, i.e.

\[
\text{put}_{j,i}^*(\text{put}_{i,j}^* (\mathcal{M}_i, h(\mathcal{M}_i, i)), h(\mathcal{M}_i, j)) \text{ is defined and}
\]

\[
\text{put}_{i,j}^*(\text{put}_{i,j}^* (\mathcal{M}_j, h(\mathcal{M}_j, i)), h(\mathcal{M}_j, j)) = \mathcal{M}_j.
\]

The coexistence of models is not sufficient for change propagation. If \( \mathcal{M}_j \) is changed to \( \mathcal{M}_j' \) by the change operation \( \text{change}_j \) then the change diagram should commute, i.e. this change operation has a related change operation \( \text{change}_i \) that allows to change \( \mathcal{M}_i \) directly to \( \mathcal{M}_i' \). The change operation is typically defined on two arguments: the original model \( \mathcal{M}_j \) and an auxiliary model \( \mathcal{M}_{aux} \). Model suite change is called synchronised for \( i, j \) and a set \( \mathcal{G}_j^{\text{change}} \) of change operations defined on \( \mathcal{M}_j \) if for each change operation \( o_j(\mathcal{M}_j, \mathcal{M}_{aux}) \) from \( \mathcal{G}_j^{\text{change}} \) a change operation \( o_i(\mathcal{M}_i, \mathcal{M}_{aux}) \) in the set \( \mathcal{G}_i \) of operations on \( \mathcal{M}_i \) exists so that the change diagram commutes for the same auxiliary model \( \mathcal{M}_{aux} \), i.e.,

\[
o_i(\mathcal{M}_i, \mathcal{M}_{aux}) = \text{put}_{i,j}^*(\text{put}_{i,j}^* (\mathcal{M}_i, h(\mathcal{M}_i, i)), h(\mathcal{M}_i, j)) \text{ for } \mathcal{M}_i' = o_i(\mathcal{M}_i, \mathcal{M}_{aux}).
\]

The complement should be constant. Therefore we may use \( h(\mathcal{M}_i, j) \) instead of \( h(o_i(\mathcal{M}_i, \mathcal{M}_{aux}), j) \).

Change operations are used for model change and evolution. The order of changes is typically important. We call two change operations \( o_i, o_j \) liberal to the models \( \mathcal{M}_i, \mathcal{M}_j \) if \( \mathcal{M}_i' = \text{put}_{j,i}^*(\mathcal{M}_j', h(\mathcal{M}_j, j)) \) and \( \mathcal{M}_j' = \text{put}_{i,j}^*(\mathcal{M}_i', h(\mathcal{M}_i, i)) \) for \( \mathcal{M}_i' = o_i(\mathcal{M}_i, \mathcal{M}_{aux}) \) and \( \mathcal{M}_j' = o_j(\mathcal{M}_j, \mathcal{M}_{aux}) \). Liberality can be extended to confluence and Church-Rosser properties [18]. Liberal change operations allow to change on model and then to apply all the changes to coherent model suites.

Contract management becomes in this case rather simple. Enforcement may directly be applied to all coexisting models. We also may restrict change operation by no action, cascade, oblige enforcements. ‘No action’ means that if a change operation cannot propagated to the other model then this change operation is rolled back. Cascade enforcement requires that the other model must be changed as well. Oblige enforcement allows to delay the change operation on the other model to a later stage.
Typically the set of change operations is defined by a change pattern. For instance, all database changes can be defined through insert, delete and update. [18] defines a small set of change pattern for extended ER language schemata that allow to express any change in HERM schemata.

Finally we may also require that the undo operation is also supported. This operation is nothing else than another change operation that results in restoring the old model.

4 Tool Support

4.1 The MetaCASE Toolkit

We introduce an approach to handle co-evolution of information systems modeling as: MetaCASE toolkits for the generation of (information) systems models for a variety of modeling languages, and model suites to assure coherence of co-existence and co-evolution of (information) system models.

MetaCASE has the capacity to generate toolkits for any modelling language. The fundamental theory behind MetaCASE is the separation of modelling concepts from their visual representations [4]. The set of concepts that can produce all expressions of a language is combined with graphic representations to function as a modelling tool. The model visualizes a graphical representation of a real world situation. According to [4] the central repository of the MetaCASE is a layered database architecture consisting of four layers: signature, language, model and data. The collection of modelling constructs (\( \mathcal{MC} \)) that allows building a set of all expressions in an arbitrary modelling language belongs to the signature layer.

The modelling constructs are elements of modelling languages. The modelling construct signature (\( \mathcal{MCS} \)) forms the hart of the signature layer. \( \mathcal{MCS} \) is a collection of modelling constructs (\( \mathcal{MC} \)), constraints or rules of modelling constructs \( \Sigma_{\mathcal{MC}} \) and a set of derivation rules for models’ population \( R(\mathcal{M}) \). This approach supports a high level of conceptuality and a sufficient comprehensibility in order to produce executable models and in order to be extendable.

Modelling constructs (\( \mathcal{MC} \)) consist of basic elements for modelling arbitrary modelling languages:

The set \( \mathcal{P} \) of roles describes predicators or associations.

The non-empty finite set \( \mathcal{OT} \) of object types contains disjoint subsets of

- label types (\( \mathcal{LT} \)), entity types (\( \mathcal{ET} \)),
- collection types (\( \mathcal{GT} \)), sequence types (\( \mathcal{ST} \)) of sets or sequences of object types, correspondingly,
- relationship types that form a partition \( F \) of the set \( \mathcal{P} \), and
- MetaCASE model types $\mathcal{MT}$ that are decomposed into modelling constructs.

**Functions:** We are given at least the following functions:
- The function $\text{Base}: \mathcal{P} \rightarrow \mathcal{OT}$ associates roles to object types.
- The function $\text{Elt}: \mathcal{GT} \cup \mathcal{ST} \rightarrow \mathcal{OT} \setminus \mathcal{LT}$ yields the elements of collection types and sequence types.

The **relation schema** defines object types for models, i.e. a subset of $\mathcal{MT} \times \mathcal{OT}$. This relation describes the elements in models.

**Specialization** $\text{Spec}$ and **generalization relations** $\text{Gen}$ as subsets of $\mathcal{ET} \times (\mathcal{OT} \setminus \mathcal{LT})$.

The **many sorted algebra** $\mathcal{D} = \langle \mathcal{D}, \mathcal{F} \rangle$ with a set of concrete domains $\mathcal{D}$ used to instantiate label types (e.g. strings, natural numbers) and a set of operations $\mathcal{F}$ (e.g. $+$).

The function $\text{Dom}: \mathcal{LT} \rightarrow \mathcal{D}$ yields the domain of label types.


To assure the functioning and operation of MetaCASE the signature layer further consist of signature constructs for storage and manipulations, versions, views, integrity and consistency, concurrency controls, security, distribution, control of integrity and interfaces. The signature constructs are orthogonal and modular components contributing to agility, extensibility and reusability. The language layer represents models of modelling languages consisting of constructs, rules and behavior normally called the (meta) models. The language layer is for the construction of (meta) models of languages and for the generation of modelling tools. This activity is called the (modelling) method engineering. The model layer accommodates language expressions; the models which
are views of the solution, and are the information systems models. The populations of model instances are in the Data layer.

4.2 Model Suite Support in MetaCASE

The main advantages of MetaCASE solutions are: the ability to configure toolkits for structure oriented modelling languages (e.g. ERD, Domain Class diagrams etc.), behavior oriented modelling languages (e.g. event charts, use cases, Petri nets, etc.), and for process oriented modelling language (e.g. Activity diagrams, Systems Sequence diagrams, DEMO etc.), the reusability, agility and extendibility to complex modelling requirements such as model suites [4].

The design, development and maintenance of complex systems such as CRMIS challenge current systems modelling by requiring a variety of tools that are capable of managing a number of changes in scope, impact, granularity, abstraction level etc. embedded in modelling support. This challenge can be met by integrating model suites into the modelling constructs of the MetaCASE. The main construction is given by:

- A model suite type $ST$ is always decomposed into MetaCASE model types $MT$.

Model suites $M_1, ..., M_n$ are sets of models with explicit associations among the models, with explicit controllers for maintenance of coherence of the models, with application schemata for their explicit maintenance and evolution, and traces for establishment of their coherence as defined in Sections 2. Thereby we achieve a holistic approach for multi-model development, management, and maintenance of information systems models.

5 Conclusion

This paper introduces the conception of model suites for support of multi-model information systems development. Model suites allow to maintain coherence of models during evolution of some of its models. Coherence is based on associations among models, controllers for maintenance of consistency within a model suite, an application schema for handling maintenance and evolution of schemata, and tracers for establishment of coherence.

The conception of model suites explicitly assumes a constructive and compositional approach to modelling. In this case, elements can be separated into
signatures, languages, models and populations (or databases). The approach is sufficient for most modelling languages such as UML diagrams, extended ER models, object-relational models, network models, or XML models.

Since the model suites concept is integrated into the MetaCASE’s signature layer, the toolkits generated within a MetaCASE in combination with the model suite concept becomes a powerful multi-model development environment. Such MetaCASE model suite toolkits can exhibit following characteristics:

1. Explicit specification of model suite collaboration: Interdependencies among models can be given in an explicit form. The consistency of models becomes recursive.
2. Integrated development of different models: Models are used to specify different views of the same problem or application. They become consistent in an integrated manner. Their integration is made explicit. Simultaneous updates of models are allowed.
3. Co-evolution of models: The model suites allow data exchange between models. The changes within one model are propagated to all dependent models.
4. Combining different representations with mathematical rigor of models: Each model consists of well-defined semantics as well as a number of representations for the display of model content. The representation and the model are tightly coupled.
5. Evolution of different representations: Changes within any model either be refinements of previous models or explicit revisions of such models. These changes are enforced for other representations as well whenever those concerns occur.
6. Management of model suites: The propagation of changes are supported by scheduling mechanisms, e.g., ordering of propagation of model changes. The management must support rollback to earlier versions of the model suite. The management should also allow model change during propagation.
7. Version handling for model suites: Model suites may have different versions.

References

Towards ASM Engineering and Modelling

Bernhard Thalheim and Peggy Schmidt

Christian-Albrechts-University Kiel, Department of Computer Science, 24098 Kiel, Germany
thalheim | pesc@is.informatik.uni-kiel.de

Abstract

The ASM approach has gained a maturity that permits the use of ASM as the foundation for all computation processes. All known models of computation can be expressed through specific abstract state machines. These models can be given in a representation independent way. Stepwise refinement supports separation of concerns during software development and will support component-based construction of systems, thus providing a foundation of new computational paradigms such as industrial programming, programming-in-the-large, and programming-in-the-world.

Despite the theoretical and application maturity a modelling theory for ASM specifications does not exist. Pragmatism and methodologies are necessary whenever larger systems have to be specified. We develop a number of principles and approaches to ASM specification that allow one to develop modular and surveyable ASMs. Our approach is based on the Turbo ASMs, abstraction layers, and on refinement. ASM engineering is based on a well-defined methodology that promises to be manageable.

1 Modelling of Applications Based on ASM

1.1 Properties of Modelling and Kinds of Abstraction

Modelling is one of the most difficult tasks in software engineering. It aims at a representation or simulation of reality and at identification of a particular model. The given application is subject to analysis by modelling if it can be described in terms of expressions in the language used for modelling. The model is a result of modelling. It relates things \( D \) under consideration with concepts \( C \). This relationship \( R \) is characterised by restrictions \( \rho \) to its applicability, by a modality \( \theta \) or rigidity of the relationship, and by the confidence \( \Psi \) in the relationship. The model is agreed upon within a group \( G \) and valid in a certain world \( W \). Stachowiak [Sta92] defines three characteristic properties of models: the mapping property (have an original), truncation property (the model lacks some of the ascriptions made to the original), and pragmatic property (the model use is only justified for particular model users, tools of investigation, and period of time). In [KT06] is additionally considered the extension property. The property allows models to represent judgments which are not observed for the originals. In computing, for example, it is often important to use executable models. Finally, the distortion property is often used for improving the physical world or for inclusion of visions of better reality.

Software engineering uses a number of principles that refine different kinds of abstraction [Tha00] such as construction abstraction, context abstraction and refinement.
abstraction. Construction abstraction uses the principles of hierarchical structuring, constructor composition, and generalisation. Refinement abstraction uses the principle of modularisation. Hierarchical structuring uses the decomposition of software into subparts in such a way that constituents form a tree. Modularisation encapsulates components and uses interfaces for exclusive communication of components with the environment. Constructor composition depends on the constructors used [Tha05]. Additionally, principles such as well-founded structuring may be applied. The last principle requires that only such constructors are applicable (sequence, bounded iteration, choice, etc.) for which the compositionality principle is preserved and semantics can be derived based on the inductive construction.

1.2 Challenges of Modern Software Engineering

Software engineering is still based on programming in the small although a number of approaches has been proposed for programming in the large. Programming in the large uses strategies for programming, is based on architectures, and constructs software from components which collaborate, are embedded into each other, or are integrated for formation of new systems. Programming constructs are then patterns or high-level programming units and languages. The next generation of programming observed nowadays is programming in the world within a collaboration of programmers and systems. It uses advanced scripting languages such as Groovy with dynamic integration of components into other components, standardisation of components with guarantees of service qualities, collaboration of components with communication, coordination and cooperation features, distribution of workload, and virtual communities. The next generation of software engineering envisioned is currently called programming by composition or construction. In this case components form the kernel technology for software and hardware.

Software development is mainly based on stepwise development from scratch. Software reuse has been considered but has never reached the maturity for application engineering. Software development is also mainly development in the small. Specifications are developed step by step, extended type by type, and normalized locally type by type. Software engineering is still be considered as handicraft work which requires the skills of an artisan. Instead, we need techniques for this century [Boe06]. Classical software development methods are mainly appropriate for programming in the small and combination of such programs into a program system. The ASM approach has the expressivity to handle also programming in the large together with programming in the small. Engineering in other disciplines has already gained the maturity for industrial development and application we need to reach.

Software engineering can be based on the trilogy consisting of the application domain description, the requirements prescriptions, and finally the systems specifications [Bjo06,Hei96]. This approach extends modern software engineering approaches by explicit consideration of the application domain.

Advanced applications such as web information systems require novel specification and development methods since their specification is oriented towards systems that are easy and intuitively to use. [Tha03,ST05] extend these approaches by (1) explicit con-
sideration of user expectations, profiles and portfolio and (2) by storyboards and story spaces.

1.3 Achievements of the ASM Approach

The ASM method nicely supports high-level design, analysis, validation and verification of computing systems:

- ASM-based specification improves industrial practice by proper orchestration of all phases of software development, by supporting a high-level modelling at any level of abstraction, and by providing a scientific and formal foundation for systems engineering. All other specification frameworks known so far only provide a loose coupling of notions, techniques, and notations used at various levels of abstraction. By using the ASM method, a system engineer can derive a general application-oriented understanding, can base the specification on a uniform algorithmic view, and can refine the model until the implementation level is achieved. The three ingredients to achieve this generality are the notion of the ASM itself, the ground model techniques, and the proper treatment of refinement.

- Abstract state machines entirely capture the four principles [ZT04] of computer science: structuring, evolution, collaboration, and abstraction.

This coverage of all principles has not been achieved in any other approach of any other discipline of computer science. Due to this coverage, the ASM method underpins computer science as a whole. We observe the following by comparing current techniques and technologies and ASM methods: ASM are running in parallel. Collaboration is currently mainly discussed at the logical or physical level. Evolution of systems is currently considered to be a hot but difficult topic. Architecture of systems has not yet been systematically developed.

- The ASM method is clearly based on a number of postulates restricting evolution of systems. For instance, sequential computation is based on the postulate of sequential time, the postulate of abstract state, and the postulate of bounded exploration of the state space. These postulates may be extended to postulates for parallel and concurrent computation, e.g., by extending the last postulate to the postulate of finite exploration.
1.4 Plan of the Paper

We are completely aware of the complexity of the modelling problem and do not expect that it can be solved within a single conference paper. Therefore, we restrict our efforts to modelling instruments of the ASM method, and to separation of concerns into development layers. In Section 2, choices for modelling are discussed: modularisation, agent orientation, styles and pattern. The section concludes with general properties. These choices are illustrated in Section 3 for a specific modelling method: layered ASM modelling. Due to space limitations we do not use sophisticated examples. We also do not discuss architectures of ASM machines. Section 4 concludes the paper with a discussion on future work.

2 ASM Modelling Alternatives

In [KT06] a general approach is proposed to modelling that starts with a clarification of the properties of modelling and of the kinds of abstraction that are considered and with an elaborated and reasoned selection of the modelling language that includes detailed knowledge of deficiencies of this language and therefore avoids the Sapir-Whorf hypothesis [Who80]. We extend this framework by an application-driven choice of architecture and platform, by a collection of modelling styles, and by orchestration of modelling techniques such as pattern. In this case, we shall be able to derive properties of the modelling process.

2.1 Modularisation

Modular modelling supports information abstraction and hiding by encouraging and facilitating the decomposition of systems [BM97] into components and their modular development based on a precise definition of interfaces and the collaboration of components through which the systems are put together. Implicit modularisation can be achieved by introduction of name spaces on signatures. Explicit modularisation offers a better understanding of structure and architecture of systems and thus supports consideration of evolution of systems and of collaboration of systems.

Modularisation offers a number of advantages: separation of concerns, discovery of basic concepts, validation and verification of development, efficiency of tool support, and - last but not least - scoped changes. The last advantage of modularisation is based on an explicit framing of development to a number of elements while preserving all other elements in its current form. We model this impact by introducing name spaces on signatures.

Typically, small submachines capture smaller models that are easier to understand and to refine. Small models can better be ascertained as to whether we need to apply refinements.

Modularization is a specification technique of structuring large specifications into modules. It is classically based on structural and functional decomposition [BS00]. We additionally consider control decomposition. Modules form a lattice of associated submachines having their own states and their own control.
Modularisation is based on implementation abstraction and on localization abstraction. Implementation abstraction selectively hides information about structures, semantics and the behavior of ASM concepts. Implementation abstraction is a generalization of encapsulation and scoping. It provides data independence through the implementation, allowing the private portion of a concept to be changed without affecting other concepts using that concept. Localization abstraction "factors out" repeating or shared patterns of concepts and functionality from individual concepts into a shared application environment. Naming is the basic mechanism for achieving localization. Parametrisation can be used for abstraction over partial object descriptions. We use the name space for handling localisation abstraction.

2.2 Agent-Oriented Specification

An ASM submachine consists of a vocabulary and a set of rules. In this case, any clustering of rules and of elements from the vocabulary may define a submachine. Turbo ASM [BS03] capture our notion of a submachine by encapsulating elements of the vocabulary and rules into an ASM. They hide the internals of subcomputations within a separate ASM. The submachine has its own local state and its own interface.

The set of functions of each submachine can be separated into basic and derived functions. Basic functions may be static functions or dynamic functions. Classically [BS03] dynamic functions can be classified as in(put) functions, out(put) functions, controlled or local functions that are hidden from the environment, and shared functions that are visible to the environment. A similar classification can also be applied to basic static functions. They are either functions only used by a its own machine or read by several environments. We thus extend the notion of shared and controlled functions to static functions as well. We do not use derived static functions since they can be considered as syntactic sugar. We differentiate these functions according to their role in Figure 2 which displays the functions internal for an agent ASM. A similar classification can be developed for functions external to an agent. An agent ASM consists of all functions that assigned to the agent and of rules that are assigned to the agent and that use only those functions assigned to the agent.

![Fig. 2. The Kinds of Internal Functions for Agent ASMs](image-url)
Static functions may also be local functions. They are not updated by any submachine. [BM97] distinguish derived functions to whether these functions are monitored functions, controlled functions, or shared functions. Typically, derived functions are functions that do not exist on their own right, but may be dynamically computed from one or more base functions. They provide a powerful and flexible information hiding mechanism. Updates made in the base functions that affect the derived function are immediately reflected in derived functions.

We may additionally assume that derived functions are allowed to update dynamic functions. In this case, dynamic functions may be used as a security mechanism, as an access mechanism, and as a simplification mechanism that allows to use complex derived functions in rules instead of complex computations in rules.

2.3 Perspectives and Styles of ASM Modelling

Different modelling perspectives can be distinguished:

1. The structure-oriented perspective focuses on structural description of the ASM. Sometimes, the structure-oriented perspective is unified with the semantic perspective. In this case, design of the structure is combined with design of invariants.

2. The behavior-oriented perspective is concerned with the behavior of the ASM during its lifetime. It can be based on event approaches or on Petri-net approaches and predicate transition systems.

3. The process-oriented perspective is concerned with the operation of the system.

The structure-oriented perspective is often used for data-intensive applications. Almost all recognized database design approaches are based on the structure-oriented perspective. The process-oriented perspective uses approaches considered in software engineering. The behavior-oriented perspective is a high-level descriptive approach to an integrated specification of the vocabulary and rules.

Modelling styles provide a very abstract description of a particular set of general characteristics of a model. Different constructional notations may be useful for describing a machine. We use the Turbo ASM approach for component or submachine description. Typically, the role of the components of the system follow the rules specified by the style. The modelling style explains the structure, the abstraction and grouping of the elements. Parts of the ASM may follow different modelling styles.

The style of modelling is a specification of the high level structure and organisation of ASM modelling. The structure describes the handling of elements of the vocabulary, the topology or relationships between elements, the semantical limitations for their usage, and the interaction mechanism between the elements such as blackboard, submodule calls, etc. The organisational style describes relevant local and global structures, the decomposition strategy, and control mechanisms between parts of the ASM machine. The organisational style is based on the architectural style. It is our aim to maintain and to preserve the strategy over the life cycle of the system.

The perspective and the style result in strategies that are use for step-wise development of specifications. The different strategies [Tha00] based on the structure-oriented perspective are sketched in Figure 3.
structure-oriented strategies

- flat (first-order) (uncontrolled) (one-dimensional)
- second-order controlled
- mixed (skeleton-based flat)
- modular (design by modules)
- inside-out (by neighborhood)

bottom-up:
1. design all basic concepts
2. build more complex concepts from them

top-down:
1. design all main concepts
2. refine concepts

Fig. 3. Structure-Oriented Specification Strategies

2.4 Pattern of ASM Vocabulary and Rule Descriptions

The notion of pattern originates from traditional architecture and denotes a general repeatable solution to a commonly occurring problem in software design. Typically, pattern are instantiatable expressions and can be transformed directly into specifications.

The vocabulary description may follow a number of different pattern. Structure-oriented strategies may apply a number of different pattern, for instance the following:

**Compacting patterns** integrate functions and represent them through one function. They provide a compact representation. For instance, the file specification in [Stä04] uses

\[
\text{argFile} : \text{isActive} \rightarrow \text{isFile}
\]

which assigns files to agents.

The compacted function

\[
\text{argName} : \text{isActive} \rightarrow \text{string}
\]

assigns a new file name to the file to be created or assigns a new name to the file to be moved. In both cases, an agent has only one task and thus is assigned to one file by \(\text{argFile} \) for each file operation. The function compacts the functions

\[
\text{argName}_{\text{Create}} : (\text{isActive} \rightarrow \text{isFile}) \rightarrow \text{string}
\]

\[
\text{argName}_{\text{Move}} : \text{isFile} \rightarrow \text{string}
\]

**Typing patterns** divide the vocabulary into types and define functions within a type or within associations among types. The vocabulary can be divided into types. For instance, the file specification is separated from the activities of agents.

**Unfolding patterns** provide all functions that are associated with a domain. In our example, all functions that are definable for agents are specified.

**Union patterns** use the most general range for functions. For instance, an agent has a main parent directory that is essentially a file. Therefore, [Stä04] specifies

\[
\text{argParent} : \text{isActive} \rightarrow \text{isFile}
\]

and tests in rules whether the result of \(\text{argParent} \) is a directory.

Each of these patterns has its advantages. Compacting and unfolding pattern allow convenient rule description. Refinement is supported by typing patterns. Unfolding pattern
lead to high redundancy that must be effectively supported. Union patterns avoid the covariance / contra variance problem but lead to problematic rules.

These pattern result in description styles. Typical description styles for structure-oriented perspective are

- predicated representation that uses a Boolean functions for the vocabulary or predicates and
- functional representation that uses functions with ranges of arbitrary domain types is appropriate for specification of event systems.

The structure-oriented perspective is typically based on a *predicative pattern* of specification.

The description of the state space can be either given based on *open world* pattern or a *closed world* pattern. The closed world pattern allows only those values that are explicitly given. The values are typically given through ground-term algebras, e.g., enumeration types. The open world pattern allows to extend the state space whenever this is necessary.

The description of terms, variable assignment, formulas, and interpretation is typically based on the *canonical specification pattern* of mathematical logics.

The rule description also may follow patterns. Typical ASM patterns are the following:

**Event-condition-action patterns** are based on a separation of the state space into event states and other states.

**Control state patterns** are based on explicit usage of control states. These states are used to separate activity of rules into those that are applicable and those that are not applicable.

**Error patterns** can be combined with any rule. They may be folded into the activities of the rule or into the condition of the rule.

**State transition patterns** are used for transition to a new state from the current state. They are typically folded into activities of rules.

**Macro patterns** are parameterised rules that allow to reuse fragments of ASM [SSB01].

We can use almost all software engineering pattern within ASM specifications. Therefore, the pattern list might be rather large. Figure 4 survey different kinds of pattern for rules.

---

![Image](image-url)

**Fig. 4.** Kind of ASM Rule Pattern

Pattern instantiation requires *fitting* of values to the parameters. This fitting is specified through context conditions. For instance, values used in conditions must fit into domains that are valid for the states potentially given for the vocabulary.
2.5 Pattern for Invariants

Invariants, e.g. integrity constraints in database applications, are used to define semantics of applications. We know different patterns for their specification:

- **Operational representation** of invariants incorporates invariants into the programs or rules. The invariant enforcement mechanism may be hidden because of control conditions or to the specification of actions.

- **Descriptive representation** uses explicit specification and refinement obligations. These descriptions are combined with the specification of invariant enforcement:
  - Eager enforcement maintains invariants based on a scheduling mechanism for maintenance of invariants. Transactional systems are typical scheduling mechanisms. They bind invariant enforcement to programs.
  - Lazy enforcement maintains invariants in a delayed mode. Inconsistency is temporarily tolerated. This tolerance reduces some of the cost of enforcing invariants within large structures.
  - Refusal enforcement maintains invariants by rollback of all activities since the last consistent state and by executing a subset of activities. Partially ordered runs are based on refusal enforcement.

Depending on the pattern chosen invariant handling is varied. If we choose an implicit invariant handling then any change applied to the current ASM must explicitly consider all invariants and must be entirely aware of the effects of these. Therefore this pattern is the most inefficient for early design phases. This pattern is however applicable during implementation if later revision is going to be based on a more general ASM.

The completeness of invariant specification is a dream that is never satisfied. Sets of invariants are inherently open since we cannot know all invariants valid in the current application, we cannot envision all possible changes in invariant sets, and we cannot choose the most appropriate selection of invariants from which all other invariants follow. Therefore, we use a separation into

- hard (or iron) invariants that must be preserved and which are valid over a long time in the application and
- soft invariants that can be preserved or are causing later corrections or which are not valid for a longer time in the application.

2.6 ASM Modelling Assumptions

The **unique name assumption** requires that elements with different names are different. If we need to use the same name for different purposes then we use **name spaces** if a unique identification is needed. The **closed world assumption** presumes that the only possible elements are those which are specified. The **domain closure assumption** limits the elements in the language to those that can be named through the vocabulary, the states of the vocabulary or the rules.

Two additional assumptions we may apply are the **unique meaning assumption** and the **universal machine assumption**. The first assumption postulates that any function or rule of the ASM has the same meaning despite modularisation. The second assumption postulates that the behaviour of the entire ASM can be defined by composition applied to the submachines.
Due to the variety of choices we might use additional assumptions for the development. The most general architectural assumption is the possibility of layering a system into sub-systems. We might use other assumptions such as common data pools, transactional systems providing an exclusive write to a location for one sub-system and a guided read with un-read to this location for all other subsystems. The use of shared functions determines whether a system consists of strictly separated components that do not have shared functions or consists of a system of components with overlapping, i.e., shared functions.

### 2.7 Properties of ASM Modelling

The software development or generally the modelling process is intentionally or explicitly ruled by a number of development strategies, development steps, and development policies. Modelling steps lead to now specifications to which quality criteria can be applied. Typical quality criteria are completeness and correctness in both the syntactical and semantical dimensions. We assume that at least these four quality criteria are taken into consideration. The modelling process can be characterised by a number of (ideal) properties:

**Monotonicity:** The modelling process is monotonic, if any change to be applied to one specification leads to a refinement. It thus reflects requirements in a better form.

**Incrementality:** A modelling process is iterative or incremental if any step applied to a specification is only based on new requirements or obligations and on the current specification.

**Finiteness:** The modelling process is finite if any quality criteria can be checked in finite time applying a finite number of checks.

**Application domain consistency:** Any specification developed corresponds to the requirements and the obligations of the application domain. The appropriateness can be validated in the application domain.

**Conservativeness:** A modelling process is conservative if any model revision that cannot be reflected already in the current specification is entirely based on changes in the requirements.

Typical matured modelling processes are at least conservative and application domain consistent. Any finite modelling process can be transformed into a process that is application domain consistent. The inversion is not valid but depends on quality criteria we apply additionally. If the modelling process is application domain consistent then it can be transformed in an incremental one if we can extract such area of change in which consistency must be enforced.

### 3 Layered ASM Modelling

We elaborate one kind of ASM modelling in more detail. Layered ASM modelling is based on modularisation and on architectures of the ASM. The language layering approach we use has already been reported in a similar form in [Wal97].
3.1 Assumptions of Layered ASM Modelling

Layered ASM modelling is based on the architectural assumption that the system can be separated into components and a general layering is achievable. We base layered ASM modelling on the unique name assumption, the domain closure assumption, and the universal machine assumption. We may use the closed world assumption and the unique meaning assumption. Layered modelling is not restricted to the last two assumptions.

In general, we use architecture-driven development that starts first with the prescription of the architecture pattern and style. The agent-oriented specification of ASM allows the development of a system as a collaborating society of sub-systems. This society uses shared functions where the sharing is based on contracts for the usage of these functions, on workflows that describe the cooperation among these sub-systems, and on implicit communication based on the locations for these functions [ST07]. We may use different views of the same architecture [Sie04] such as technical views displaying the modules with their functionality, application views displaying activity zones depending on the stage of the application, infrastructure views displaying the dependence of the system from its infrastructure and supporting systems, or the context view that considers the whole organisational, story and application context.

3.2 Vocabulary Modelling

The five properties of modelling are: mapping, truncation, pragmatic, extension and distortion properties. These properties govern abstraction. We consider six different aspects for vocabulary modelling: intention, usage, content, functionality, context, and realisation. The intention aspect is a very general one centered around a mission statement for the system. The primary question is: what is the purpose of the system? Once some clarity with respect to the intentions of the system has been obtained, it is important to anticipate the behaviour of the users. The content aspect concerns the question: Which information should be provided? The functionality aspect is coupled with the question, whether the system should be passive or active. The context aspect deals with the context of the system with respect to society, to time, to expected users, to the history of utilisation and to the paths of these users through the system. The realisation aspect concerns the final implementation.

Vocabulary modelling must cover all these aspects in a proper form. It may be based on partial order-sorted signatures. Depending on the choices we made for modularisation, separation of concerns through agent-oriented specification, perspectives and styles, pattern, and properties of modelling itself we can use different representations and conventions. Typical conventions are naming and binding conventions.

Layered ASM modelling can be based on typing pattern. We are additionally interested in incrementality and application domain consistency. We additionally assume value-identifiability for each element of the state. This assumption allows to define equality and inequality for each pair of elements of the state. Therefore we develop a multi-layered vocabulary modelling approach that uses the following ingredients:

**Domain types** are used for introducing the set of states envisioned. The superuniverse is the union of all states. Beside basic value types such as string, int, float etc. we assume domain types
BOOL, NULL, Ø, ID

consisting of the truth values true, false, of the value undef, of the empty set, and of a set of identifiers, correspondingly.

Domain types can be complex types that are inductively constructed from basic types by applying constructors such as (Cartesian) product, set, list, and multi-set constructors.

Domain types can be labelled by an abstract name. Domain types are typically order-sorted. We restrict the partial order to lattices or Brouwerian algebras.

Abstract types are denoted by a triple consisting of an abstract name denoting the abstract type, of a domain type used for values, and a set of invariants limiting the state space that can be used for interpreting the abstract type. The set of invariants may be empty. In this case we omit the third part of the triple. If the abstract name coincides with the domain type name then it can be omitted as well.

Predicates are specified by their name, an arity, a sequence of abstract types, and a set of invariants, where the length of the sequence coincides with the arity. Invariants limit the state space that can be used for interpreting the predicate. It can be empty. Predicates are interpreted on the basis of set semantics.

Functions are specified by their name, an arity, a sequence of abstract types used for the domain of the function, an abstract type used for the range of the function, and a set of invariants where the length of the sequence coincides with the arity. Invariants limit the state space that can be used for interpreting the function. It can be empty. Functions are interpreted on the basis of set semantics for their domain.

Predicates and functions are often partial. Due to application domain consistency we assume that each abstract type has a natural meaning in the application domain. We also may assume that labels have a natural meaning in the application domain or are used as an abstraction for convenience of specification. We use the unique name assumption for all labels, types, predicates and functions. It is convenient to assume that abstract names and the superuniverse are disjoint.

Additionally we support the introduction of derived notions:

View functions are derived functions. They can be virtual or materialized. Virtual view functions are computed whenever this is necessary. They are not stored in the ASM. View functions can be given in

- an explicit form by their introduction in the vocabulary and
- an implicit form by their introduction in rules using the let, choose, for all and where introductions. We may distinguish

  • transient view functions that are used for definition of values in let and choose rules and may be directly removed whenever a value has been chosen and
  • collector view functions that are used for parallel execution in forall rules and in where introduction and must have a lifespan that last over the entire rule computation.

Virtual view functions are the view functions typically used. In this case, it it assumed that the view is computed first before applying a rule, e.g. in a where introduction.
Clusters functions are disjoint unions of functions. They are well known in programming languages and are mainly used as syntactic sugar since generalisation of functions by combination eases the treatment.

Control state functions are often introduced in rules through conditions that use a certain control state as the enabler or disabler of the rule.

Special purpose functions are used for default functions and exception functions. Each of the elements of the vocabulary has its scope that consists of the space of all locations that are used for the functions and of all rules that use these elements.

All these derived notions may be used inductively for the construction of other derived functions. Therefore, the vocabulary becomes layered depending on the construction.

Additionally, we need to consider metadata describing the specific purpose of elements of the vocabulary. They represent the content and the meaning of the functions. The meaning can be partially described by the name if the wording used has a mini-semantics. Metadata also include technical data that guide refinements applied to the vocabulary. Metadata may be partially given through a glossary or thesaurus.

Finally, a well-developed specification uses naming conventions for readability.

We distinguish between:

- sketchy specification of the vocabulary that uses an implicit definition of domain types and of abstract types and that specifies predicates and functions through declaration of the mappings and
- sophisticated specifications of the vocabulary with detailed description of any element of the vocabulary.

The typical specification of the vocabulary uses a specification approach between these extremes. Whenever refinements need to be made then sophistication is necessary for the verification of refinement correctness.

3.3 State Space Modelling

State and vocabulary are two sides of the same coin and must be developed in a co-design process. We distinguish two extremes:

- Orientation to most general domain types: We use the most general domains for the specification. If we need specific domains then we can use unary predicates. The exclusive utilisation of most general domain types leads to complex invariants for values with specific meaning and for relations among the values and to extensive specification of various ‘exceptions’. The refinement is simpler but the burden for correct functioning is transferred to rule specification.

- Tight coupling of domain types and abstract types: Any abstract type is associated with the most specialised domain type. Specialised domain types are often associated to a specific meaning in the application. Refinement becomes more difficult. At the same time, rule specification can concentrate on the essentials. Exception due to weak domain types do not appear.

Enumeration states are often used for control states. These control states separate transitions and lead to implicit clustering of rules. Control states have a implicit scope that
is defined by their utilisation in conditions and by transfer assignments in rules from or to another control state. Rules that are enabled by a control state form a submachine or a module. The transfer from one control state to another control state is represented by a state transition graph.

### 3.4 Agent-Based Modularisation

We combine modularisation and agent-oriented specification. Each function and each predicate may be visible or may not visible to an agent. Any agent has its *vocabulary share*. This vocabulary share states whether the given predicate or function is an in element (a monitored function or predicate), a controlled element, a shared element or an out element. A function or predicate may also be external (denoted by \( E \)) for an agent.

Therefore, given a set \( A \) of agents and a vocabulary \( V \), the vocabulary share is given by a function

\[
\text{share} : A \times V \rightarrow \{E, I, O, C, S\}
\]

assigning agents their kind of vocabulary use.

If an agent uses a dynamic function or predicate as its controlled function then no other agent can use it. If an agent uses a dynamic function or predicate as its in or out element then we should require that another agent uses this function or predicate as an out or an in element, correspondingly. We may additionally require that dynamic functions and predicates of the vocabulary are partitioned into in, out, controlled, and shared functions. Typically, we require that the shared dynamic function predicate are consistently assigned, i.e., the function or predicate is either external or shared for any agent that uses it internally. The same restriction can be made for static functions. This restriction is a variant of the well-known open-closed principle [Mey88] for vocabulary in layered ASM modelling.

According to Figure 2 we restrict static functions and predicates to controlled and shared share. We also exclude using derived functions and predicates as indirectly out functions.

Derived notions may be restricted to be used only as means for the simplification of vocabulary definition or as services provided by agents. In the later case they will have the same behaviour as out functions and predicates. They may be used by other agents.

It is sometimes convenient to use the elements of the vocabulary in a mixed form. For instance, a dynamic function is an in function for one agent and a shared function for other agents. We avoid this approach for layered ASM modelling since it can be a source for confusion.

Layered ASM modelling also aims to establish a clear definition of the kind of sharing. We introduce an abstract type *right* that denotes the rights an agent may have for updating the shared elements and an abstract type *obligation* that denotes the obligations an agent must follow if this element is a shared element for the agent. We introduce the contract for an agent by assigning rights and obligations to agent for each shared function or predicate by a partial function

\[
\text{contract} : A \times V \rightarrow \text{right} \times \text{obligation}
\]
that assigns the right and the obligation to each agent for each function or predicate the agent shares. It is often convenient to use the contract function as a dynamic function that can be changed by an agent that acts in the role of a controller or scheduler. In this case we can assign to an agent an exclusive update right while excluding other agents from writing. These agents have in this case only read rights. This approach eases introduction of transactional systems.

Additionally, we may use a general agent \texttt{Main} for the main ASM.

### 3.5 Modular Rule Modelling

Rules are mainly specified based on different condition-action patterns, e.g., event-condition-action pattern, control state pattern, state transition pattern. The basic specification of a rule is extended by

- **state dependence** that describes whether a rule can be invoked in dependence of certain conditions on a state,
- **access environment** that determines where the invocation of a rule may appear, and
- **control guards** that restrict invocation of a rule and are handled by a controller ASM.

The extension has been introduced for convenience. It can be expressed by an ASM that is generically added to the given ASM. We also use this extension for explicit treatment of conflicts in update sets of rules. This extension does not limit the application of partial ordered runs but allow an explicit treatment of such updateset conflicts that must be treated.

State dependence condenses conditions on the existence of certain objects in the state, value conditions for the invocation of a rule, and collaboration conditions that relate the given rule to the invocation of other rules. Collaboration conditions can also be used for explicit synchronisation of parallel rule invocation. State dependence specification supports subject-oriented programming that focuses on capturing different subjective perspectives on a single object model. It basically allows composing applications out of “subjects” (partial object models) by means of declarative composition rules.

The access environment specification contains the views on a state that are initialised when a rule is going to be executed and the internal functions that are used for execution of the rule. Each of the rules has its scope that consists of the space of all functions and predicates that are used in the rule. The scope contains all parameters of the rule. The scope can be used to bind an element of the vocabulary to all rules have this element within its scope. This inversion is called vocabulary element scope.

The open-closed principle for rules in layered ASM modelling: *If an agent is assigned to a rule then the scope must only contain such elements of the vocabulary on which the agent has an internal share.*

Control guards allow one to avoid inconsistent update sets. We use a partition of control states for separation of runs of abstract state machines. Control guards may be used as entry guards that restrict invocation of a rule and accept guards that restrict updates of a rule. The allow one to express rely-conditions and guarantee-conditions by
both pre- and post-conditions. Rely conditions state what can be tolerated by the party. Guarantee-conditions record the interference that other processes will have to denote with if they are allowed to run in parallel. We envision in this paper that these conditions can be generalized to a specific style of assumption-commitment specification. [Wal97] and [SSB01] use exception types which are a very specific type of control guards. Control guards may also be used for introduction of break and continue statements for a rule with temporary lock and wait views.

Rules may be combined to form macros, whichmay be reused by other submachines. Macros support intentional programming and aspect-oriented separation of functionality. They can be generalised for adaptive programming, generative programming, and pattern-based development. Intentional programming provides an extendible programming environment based on transformation technology and direct manipulation of active program representations. New programming notations and transformations can be distributed and used as plug-ins in a play-in/play-out engine [HM03]. Aspect-oriented programming improves the modularity of designs and implementations by allowing a better encapsulation of cross-cutting concerns such as distributed transfer, synchronization, data traversal, tracing, caching, etc. in a new kind of modularity called ‘aspects’. Generative programming aims to increase the productivity, quality, and time-to-market in software development thanks to the deployment of both standard component and production automation. System families are developed rather than single systems. Generative programming uses government and binding [BST06].

4 Concluding: Future Plans for Evolution of ASM Modelling

4.1 Treatment of Over-Specification

[HT00] discussed the downside of over-specification. A good specification contains a protocol for future extension and a portfolio for the current implementation. It is a contract with the application stakeholder and an unambiguous description of the application domain. Therefore, it seems that the specification must be as complete as only possible. This ‘completeness’ leads to the over-specification. Formal methods should not be misused for a hyper-detailed description of an application but should provide robustness against other interpretations and understandings, against changes in the application itself or within the computing environment, against evolution of the application domain, and against multiple styles of the specification. A hyper-detailed specification suffers from the straitjacket effect that limits the flexibility of specification.

The ASM methods offers executability of the specification and thus treats the strait-jacket effects and supports robustness. The over-specification problem can however been solved by explicit introduction of checkpoints that allow to overcome the dangers of over-specification. A check point measures the specification itself. These measures might be build in a similar form as metrics. One possible measure could be the vocabulary complexity of rules. We might use a threshold value which should not be exceeded for any rules. This threshold value can be based on the scope of the rules.
4.2 Deriving Plans and Primitives for Refinement

The perspectives and styles of modelling rule the kind of refinement styles. As an example we consider structure-oriented strategies of development depicted in Figure 3:

**Inside-out refinement:** Inside-out refinement uses the given ASM machine for extending it by additional part. These parts are hocked onto the current specification without changing it.

**Top-down refinement:** Top-down refinement uses decomposition of functions in the vocabulary and refinement of rules. Additionally, the ASM may be extended by functions and rules that havenot yet been considered.

**Bottom-up refinement:** Bottom-up refinement uses composition and generalisation of functions and of rules to more general or complex ones. Bottom-up refinement also uses generation of new functions and rules that have not yet been considered.

**Modular refinement:** Modular refinement is based on parquetting of applications and separation of concern. Refinement is only applied to one module and does not affect others. Modules may also be decomposed.

**Mixed skeleton-driven refinement:** Mixed refinement is a combination of refinement techniques. It uses a skeleton of the application or a draft of the architecture. This draft is used for deriving plans for refinement. Each component or module is developed on its own based on top-down or bottom-up refinement.

These different kinds of refinement styles allow one to derive plans for refinement and primitives for refinement.

4.3 Generic Refinement Steps and Their Correctness

[Bör03,Sch05] have developed a general theory to refinement. Control of correctness of refinement takes into account (a) a notion of refined state and refined vocabulary, (b) a restriction to states of interest, (c) abstract computation segments, (d) a description of locations of interest, and (e) an equivalence relation among those states of interest. The theory developed in [Bör03,Sch05] allows to check whether a given refinement is correct or not.

A typical engineering approach to development of work products such as programs or specifications is based on a general methodology, operations for specification evolution, and a specification of restrictions to the modelling itself. Each evolution step must either be correct according to some correctness criterion or must lead to obligations that can be used for later correction of the specification. The correctness of a refinement step is defined in terms of two given ASM together with the equivalence relations. Already in [Sch05] it has observed that refinement steps can be governed by contracts. We may consider a number of governments [BST06] in the sense of [Cho82]. However we should take into account the choices for style and perspectives.

Given a refinement pattern, perspectives, styles and contract, we may derive generic refinement steps such as data refinement, purely incremental refinement, submachine refinement, and (m,n) refinement. The generic refinement is adapted to the assumptions made for the given application and to consistency conditions. Typically such consistency are binding conditions of rules to state and vocabulary through the scope of rules. The general approach we envision is depicted in Figure 5.
We are currently developing a number of refinement steps that take preconditions for their enactment and use postconditions for their deployment. The derivation of pre- and postconditions and of is based on principles used for government and binding.

References


Remark: This research proposal is an answer to an email exchange between Daniel Klünder and Andreas Prinz who summarised: “Engineering or modelling of ASM itself has not yet given the right attention.” This paper attempts in development of a general ASM modelling approach.

Acknowledgement: We are very thankful to our reviewers and S. Hegner for their detailed, fruitful and challenging proposals, remarks and requests.
ASM-Supported Management of UML Clusters

Peggy Schmidt and Bernhard Thalheim

Christian-Albrechts-University Kiel, Computer Science Institute, 24098 Kiel, Germany
pesc|thalheim@is.informatik.uni-kiel.de

Abstract

Software engineering starts with informal descriptions of the application domain and results in software that satisfies a number of properties. The main target of software engineering is the management of the development process. Current software engineering practices do not succeed in this regard. UML diagrams are ubiquitous in software engineering, forming the cornerstones of modelling techniques. A large variety of UML diagrams is used for specification of different aspects of the application. Their integration is either done on intuition of the developers or partially based on OCL constraints. While the lack of semantics in UML is a problem, the paper does deal with sets of UML diagrams for management and organisation of such diagrams and their semantics into clusters of UML diagrams. The entire process is backed by abstract state machine (ASM) specifications. Consistency of different views and of evolution of software engineering products is based on contracts. UML diagrams are considered to be views on the ASM specification and have thus a well-defined semantics. At the same time we may provide also a methodology for the development of such diagram sets. We will show an idea of an automatic transition from the a set of UML diagrams, called UML diagram clusters, to the machine-readable ASMSpec. These transformations may also be written in ASM. So we support a precise meaning of these UML diagram clusters through the ASM-based generation of ASM specifications. The generated ASM specification is easy and intuitive to understand so users are able to understand what the concrete UML diagram cluster means. The consistency of a UML diagram cluster and between diagram clusters is supported through ASM-based contracts.

1 Introduction

Research has been conducted in software engineering for more than thirty 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 consists of computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system. 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 [?]. Software engineering should thus be a systematic approach to the development, operation, maintenance, and retirement of software.

UML diagrams are ubiquitous in software engineering, forming the cornerstones of modelling techniques. They became the most widely used tools for software specification and a de facto standard. This approach results in a gap:

Problem 1. 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.

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 2. 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 3. Evolution of different representations: Changes within any UML document, called UML diagram diagram clusters 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.

Theses are a number of distinct formalizations of the semantics of UML diagrams. This makes the management of UML a mental challenge in the context of team development and the specification of large projects. Moreover, the quality
of software engineering products is given mainly in an informal way. Quality characteristics such as ISO/IEC 9126 are given in a very informal way and are mainly based on counting code chunks that obey certain properties.

**Problem 4. Adding mathematical rigor to representations:** Each work product 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.

### 1.2 ASM-Based Software Engineering

Abstract state machines 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 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 seen or understood as a behaviour of the other machine.

The ASM approach may be used to solve the four problems of software engineering:

1. **Adding mathematical rigor to representations:** UML diagrams have a partially defined semantics and leave freedom of interpretation to developers. This freedom may however cause communication problems and results often in complete confusion. We may use another approach. Each UML diagram is bound to a set of UML diagram clusters. Each UML diagram cluster is bound to an ASM specification. The freedom of interpretation is given by drivers and transformations that allow one to adapt this corresponding to a specific interpretation style. UML diagrams thus provide a precise mathematical meaning. This meaning depends on the choice of drivers and transformations we have chosen for clarification of the missing part of semantics.

2. **Integrated development of different representations:** Instead of managing a large number of representations in a bilateral mode we can use the ASM specification as the semantics of the UML diagram cluster. As long as any change within a UML diagram cluster can be mapped to a change in all corresponding UML diagrams clusters we are able to maintain consistency among these clusters.

3. **Flexible change management:** Changes in some of the UML diagram clusters are bound by contracts. Changes that are bound by contracts are symmetric (i.e. not biased to some diagrams), local (i.e. localise changes to the update scope of the diagram cluster), and extensible (i.e allow to be changed themselves the contracts).
4. Evolution of different representations: Each of the UML diagram clusters that is currently under consideration is a coherent view of the ASM specification. Any change applied to one of the UML diagram clusters can be thus considered to be a refinement of the previous views for the previous ASM specification.

Large software systems can be simplified tremendously if techniques of modular modeling such as design by UML diagram clusters are used. Modular modeling is an abstraction technique based on principles of hiding and encapsulation. Design by UML diagram clusters and its corresponding ASMSpec 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.

1.3 Organisation of the Paper

The paper is organized as follows. Section 2 introduces the generation of ASM specifications on the basis of UML diagrams over diagram clusters and the role of ASM based contracts to manage UML diagrams. Section 3 discusses an example within our approach. We give examples for the transformation UML-Diagrams into ASM specifications and for the check of consistency between UML clusters. Section 4 summarizes the paper and discusses the approach of software engineering backed by ASM.

2 Managed Engineering of UML Diagram Clusters Based on ASM

The software-development process yields a partially ordered set of UML models that should be consistent. The semantics of the UML diagram cluster are should be precisely defined. The problem thus arises of how to understand and how to check consistency between diagrams of a UML diagram cluster and between UML diagram clusters.

2.1 UML Diagram and Diagram Type

Software engineering with graphical tools like UML, assumes that a diagram type is defined before any diagram can be modelled. The role of the diagram type is to specify specific possible structures of the diagram that will be used to specify a view of an application. A diagram type defines uniform structure and constraints on diagrams.

Let an underlying diagram type system be defined as

\[ t = b | (a_1 : t_1; \ldots ; a_n : t_n) | \{ t \} | [t] \]
Here b represents an arbitrary collection of base diagram types
the constructors
• \((\ldots)\) are used for records (or diagrams),
• \(\{\ldots\}\) for finite sets, and
• \([\ldots]\) for finite lists.

A diagram type of level \(k\) has a name \(\mathcal{E}\) and consists of a set
\[\text{comp}\mathcal{E} = \{r_1 : E_1; \ldots; r_n : E_n\}\]
of components with pairwise different role names \(r_i\) and diagram types (or clusters) \(E_i\) on levels lower than \(k\) with at least one diagram type of level exactly \(k - 1\),
a set \(\text{attr}(E) = \{a_1, \ldots, a_m\}\) of attributes, each associated with a basis diagram type \(\text{dom}(a_i)\) as its domain, and a key \(\text{id}(E) \subseteq \text{comp}(E) \cup \text{attr}(E)\).

We shall write \(\mathcal{E} = (\text{comp}(\mathcal{E}), \text{attr}(\mathcal{E}), \text{id}(\mathcal{E}))\).

A cluster of level \(k\) has a name \(\mathcal{E}\) and consists of a set
\[\text{frag}(\mathcal{E}) = \{f_1 : \mathcal{E}_1; \ldots; f_n : \mathcal{E}_n\}\]
of fragments with pairwise different fragment names \(f_i\) and diagram types (or clusters) \(\mathcal{E}_i\) on levels at most \(k\) with at least one of the \(\mathcal{E}_i\) of level exactly \(k\).

A "specification tool" an S-indexed family \(\{\text{D}(\mathcal{E})\}_{\mathcal{E} \in S}\) of finite sets \(\text{S}\) of diagram types and clusters such that for all \(\mathcal{E} \in S\) and all \((r_i : \mathcal{E}_i) \in \text{comp}(\mathcal{E})\) or \((f_i : \mathcal{E}_i) \in \text{frag}(\mathcal{E})\), respectively, we also have \(\mathcal{E}_i \in S\).

A diagram over a diagram type \(\mathcal{E}_i = (\text{comp}(\mathcal{E}), \text{attr}(\mathcal{E}), \text{id}(\mathcal{E}))\) is a partial mapping \(t\) defined on \((\text{comp}(\mathcal{E}) \cup \text{attr}(\mathcal{E}))\) such that the following conditions hold:

- \(t(r_i : \mathcal{E}_i)\) is either undefined or a diagram over \(\mathcal{E}_i(\text{for } r_i : \mathcal{E}_i) \in \text{comp}(\mathcal{E})\);
- \(t(a_j)\) is either undefined or a value in \(\text{dom}(a_j)\) (for \(a_j \in \text{attr}(\mathcal{E})\));
- \(t\) is always defined on all element of the key \(\text{id}(\mathcal{E})\).

A diagram \(\mathcal{C}\) over a cluster \(\mathcal{E}\) is a pair \((f_i : t_i)\) for some \((f_i : \mathcal{E}_i) \in \text{frag}(\mathcal{E})\) and a diagram \(t_i\) over \(\mathcal{E}_i\). A system specification \(\mathcal{D}\) over a system specification \(S\) is an S-indexed family \(\{\text{D}(\mathcal{E})\}_{\mathcal{E} \in S}\) of finite sets \(\text{D}(\mathcal{E})\) of diagrams over \(\mathcal{E}\) such that the following conditions hold:

- If \(\mathcal{E}\) is a type, then for each \((r_i : \mathcal{E}_i) \in \text{comp}(\mathcal{E})\) and each \(t \in D(\mathcal{E})\) for which \(t(r_i : \mathcal{E}_i)\) is defined \(t(r_i : \mathcal{E}_i) \in D(\mathcal{E})\) holds;
- If \(\mathcal{E}\) is a cluster, then for each \((f_i : \mathcal{E}_i) \in \text{frag}(\mathcal{E})\) and each \((f_i : t_i) \in D(\mathcal{E})\)
  we have \(t_i \in D(\mathcal{E}_i)\).

The diagram type \(\triangledown \mathcal{T}\) of any UML diagram \(\mathcal{T}\) specifies its abstract syntax, and provides guidelines for creating an UML diagram at type level. A diagram type \(\triangledown \mathcal{T}\) can instantiated to diagrams.

We use diagram types to decribe the syntax of diagram diagrams, while semantics details are expressed in ASM-based semantic transformations uml2asm.

Thus, we now define a UML diagram clusters \(\mathcal{C}\) are called formal if written in a notation that already has a precise meaning. So we need fo each UML diagram
clusters a semantic. This semantic has to be coded in the transformation rules in to a ASM Specification.

The UML cluster requires additional details and depends on the syntax of the UML cluster. So we have to define equivalence relations defined on UML diagram clusters which may disregard such details, and yet provide satisfactory notions of semantic equivalence and implementation correctness. The formal nature of this semantics is adequate for justifying formal analysis methods for UML diagrams. Given a syntactically correct UML diagram cluster, a static semantics is needed in order to determine whether the UML diagram cluster is well-formed, and thus the ASMSpec is executable. The semantics of the executable ASM then provides a model of the ASM execution. Thus, with the help of ASM we get a dynamic semantics for the UML diagram cluster. The Semantics of UML diagram clusters can be viewed as a function that interprets well-formed formulae UML diagrams in a semantical domain, called the UML diagram cluster transformation. A UML diagram cluster \( C' = (C, CM, \text{uml2asm}) \) contains a diagram \( C \), a coherence matrix \( CM \) and ASM-based semantic-transformations \( \text{SemT} \). A UML diagram cluster \( C' \) has to conform to its cluster type \( E \).

We assume a set of elements \( e \in C \) denoted by \( M \). A pair in \( M \) refers to an UML elements pair \( \{e_1, e_2\} \), where \( e_1, e_2 \in M \). A coherence matrix \( CM \) describes the set of pairwise relationships, between UML elements \( M \) of a UML diagram cluster. It is a mapping \( c \) from \( M \times M \) to a boolean value. A coherence matrix is reflectiv \( (\forall a : c(a, a) = 0) \) and symmetric: \( \forall a, b : c(a, b) = M(b, a) \). The morphism \( \text{uml2asm} : CM \times ASMSpec \rightarrow ASMSpec \) assigns to a coherence matrix \( CM \) and a chosen ASM-based transformation an ASMSpec. The transformation may use a variety of transformation styles. If the transformed \( C \) cannot be embedded into an ASMSpec then the result is \( \bot \). Each element \( M \) has a role in the transformation process from a UML diagram cluster to an ASM specification ASMSpec, which models a high-level specification for a piece of an application.

**Saturatedness:** A mapping set for a UML diagram cluster \( C \) is saturated if each required element \( E \) of the mapping set \( \text{sat} \) from the \( C \) to ASMSpec is available.

**Consistency:** A mapping set for an instance of UML diagram cluster \( ucd \) is consistent if each constraint \( c \) of the mapping set \( \text{cons} \) from the \( C \) to ASMSpec is satisfied.

As the UML diagram cluster may consist of several UML diagrams each of which is a reusable component in and by itself, each UML diagram is assigned a unique namespace to avoid naming conflicts.

UML diagram cluster unification is the operation of combining two UML diagram clusters \( C \) so that the result is the most general coherence matrix \( GCM \) that is subsumed by the two unificands. If there is no such structure, then the unification fails. Two \( C \) that can be unified are compatible (or consistent). If two \( C \) are comparable then they are compatible.

We are now able to use the general coherence matrix \( GCM \) in two ways.
1. A new UML diagram cluster \( C = C'1 + C'2 \) is obtained by interchanging the general coherence matrix \( GC\). If the UML diagram cluster type \( CT \) of \( C'1 \) and \( C'2 \) is satisfied.

2. We can use \( C'1 \) and \( C'2 \) in a collaboration by interchanging the general coherence matrix \( GC\).

2.2 Contracts for UML Diagram Clusters and ASM Specifications

Contracts restrict UML diagram clusters to those that are considered as legal. We establish an interface that is based on contracts which bind views to the UML diagram cluster and to the ASM specification. Contracts are fundamental mechanism for UML diagram collaborations and ASM specifications of the UML-cluster integration. Contracts (in the sense of our approach) are based on \( C \)'s. A development contract performs services inside one \( C \). A collaboration contract performs rules for the modification behaviour of two \( C \).

Constraint enforcement occurs when an update, insertion, or deletion is executed within the UML diagram cluster (or ASM specification). All changes must preserve saturatedness and consistency after firing an update operation. The contracts may use different norms and types. Contracts follow a number of norms that are given by rules, regulations or agreements among the parties involved. A contract provides the following information:

- roles involved, called parties;
- relationships between contracts;
- the period of service (begin and end of contract);
- the status of contracts;
- a contract monitoring facility that performs checking of the fulfillment of obligations and the monitoring of constraint satisfaction;
- a contract notification component that sends various contract notifications to the parties;
- other components and facilities to support contract negotiations, enforcement and also dynamic configurations of the system to reflect new rules and structures.

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

**Registration** In the registration phase, two agents are involved 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 ASM-cluster under the supervision of the contract manager. 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 applies to UML diagrams or ASM specifications. We distinguish obligations for the acting and the reacting agent, permissions given by the reacting agent to the acting agent, and sanctions applied by the supervisor 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 contract manager. 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 agent must satisfy before using the service as well as the constraints that are guaranteed by the server when used. The **Activation** defines preconditions for obligations, permissions or sanctions. **Finalization** assumes the **activation** is true. If the **activation** and **finalization** conditions are met then the service **permissions** must be preserved.

We separate collaboration contracts services into simple modifications and composite modifications. Simple modifications fulfill atomic changes. Composite modifications represent bundles of changing operators that are utilized together. A composite modifications depicts an nested transaction. The contract has to satisfy the conditions for the UML diagram cluster after the applied modification. The development contract specifies which modifications can be applied in the UML diagram cluster. The constraint states that the modification is applicable only if the condition \(|\frac{C}{\} |\) is true.

The collaboration contract specifies the interaction between UML diagram clusters and how to propagate these modification to the affected UML diagram clusters. The contract states what conditions after the execution of the modification have to be true. When a UML diagram cluster instance is used, all the requirements should be satisfied and all the elements have to be bound to elements of the other UML diagram cluster, that means that the coherence matrix has to be modificate accordingly. The consequence of one operation could be the involvement of a sequence of contracts over UML diagram clusters. Contracts could define the ACID or customised nested transaction, including a fine grained abort management for executions of a service.

A modification contract can be instantiated if two \(C\) are comparable. The modification contract of two \(C\) are based on a subset their unification. We separate \(C\) which are type identical or token identical. Two \(C\) are type identical if they are of the same type. They are consistent if their type values are consistent. Their general coherence matrices \(GCM\) have consistent values. Two \(C\) are token identical if they share the same general coherence matrices \(GCM\). Two \(C\) are consistent if they have the same value, the values of their general coherence matrix \(GCM\) are consistent.
The generation rules of contract templates is based on ideas of “two-level grammars” \[?]\ The contract generator translates generation rules into a contract template ASM. This contract template is saved in the dynamic function \texttt{template}. The set of formal parameters consists a set of values of dynamic function and the rules are the methods that manipulate the instance variables. We call the first level of generation analogously to \[?]\ meta rules of a contract templates, while the second level parameterized is called hyper-rules. In Figure ?? is shown a sketch of ASM specification which generates contract templates.

\begin{figure}
\begin{verbatim}
Norm = \{Obligation, Permission, Sanction\}
nome : Contractpattern → String
agent : Contractpattern → 2Agent \cup 2Role
norms : Contractpattern → 2Norm
Normtype : Norm → Norm
name : Norm → String
activation : Norm → Condition
expression : Norm → Action \cup Condition
finalization : Norm → Condition
role, name : Expression → Role

Generate\(ct\) =
template\(ct\):= template\(ct\) + 'Contract' + name\(ct\)
\textbf{choose} a ∈ agent\(ct\) \textbf{do}
 template\(ct\):= ' IF role(a)==' + role\(ct\) ;
agent\(ct\):=agent\(ct\) - a ;
\textbf{choose} n ∈ norms\(ct\) \textbf{do}
 template\(ct\):= template\(ct\) + type(n) + ' ' + name(n) + () ;
template\(ct\):= template\(ct\) + ' IF event(c) == ' + name(n) then ' ;
 Activation\(ct, n\);
 Expression\(ct, n\);
 Finalization\(ct, n\);
 norms\(ct\):=norms\(ct\) - n ;
 Activation\(ct, n\) =
template\(ct\):= template\(ct\) + activation(n) ;
template\(ct\):= template\(ct\) + ' state(c,type(n) + ' + name(n) + ') := activated ' ;
Expression\(ct, n\) = template\(ct\):= template\(ct\) + expression(n) ;
 Finalization\(ct, n\) =
template\(ct\):= template\(ct\) + finalization(n); template\(ct\):= template\(ct\) + ' state(c,type(n) + ' + name(n) + ') := completed ' ;
\end{verbatim}
\caption{Contract pattern generator}
\end{figure}


3 Example: Library Support System

The approach taken in this paper is based on contracts and on the local-as-view approach. We use synchronization for consistency management and UML diagram clusters as views. UML diagrams themselves and UML diagram clusters have various interpretations, different semantics for different domains.

The approach depicted in Figure ?? integrates the UML diagram cluster based on an ASM specification. The code is automatically generated on the basis of a chosen transformer and evolves during the development process. The UML diagrams are bound to the ASM specification. This contract regulates in which case which changes can be applied, and which must be applied or are forbidden to or for existing UML diagrams or to UML diagrams generated from the ASM specification. UML diagram clusters partially depend on each other. Our approach uses integration by design-by-contracts and results in a method for consistency management of UML diagrams.

Specification clusters are supported by different refinements or revisions. We use for ASM specification the approach taken in [?]. We may use the general syntax of UML diagrams, for example, similar to [?] and [?]. The evolution of UML diagram clusters can be mapped to contracts, which are transition constraints that maintain properties on UML diagram clusters. The satisfaction of these contracts is maintained by the contract manager.

3.1 Demonstration of Change Management

We demonstrate the approach on the basis of a small example for a library support system that shows how stepwise co-evolution of diagrams can be performed.

We start with a description by use cases and state-charts (behavioral diagrams). These diagrams can be mapped to an ASM specification. Any change applied to any of these diagrams is only applicable if the contracts among the diagrams and the ASM specification are preserved. As a side effect these UML diagrams are treated as integrated diagrams and can co-evolve.

We use a contract manager for management of deployment of the UML diagram cluster and ASM Specifications based on three services: generating a ASM specification from a UML diagram cluster, decomposition of classes, and collaboration of two UML diagram clusters.

In the example two UML diagram clusters \(C'1\) and \(C'2\) are depicted. A use case is a set of scenarios. Each scenario describes a sequence of events. We explicitly attach actors and roles to the events. The use case in Figure 2 shows two units of functionality (borrow and return a book from the library) provided by the system. The goal of this use-case diagram is to help development teams visualize the functional requirements of a system, including the relationship of "actors" (employee and student) to relevant processes. Figure 8 depicts user added ASM code after automated creation of these use cases.

\(C'1\) (see Figure 2) is based on the standard class diagram type \(\triangle T\) and contains the class diagram "class1", which illustrates the static structures of the system, that means how different entities (i.e., the book, the person, the
borrowedBook) relate to each other. To model the use cases and the static structure is the task of author Author1.

UML Statechart diagrams focus on the event-ordered behavior of an object, and present the event-triggered flow of control due to transitions which lead from state to state. A dynamic function $ctl\_state$ is generated. The domain of this function gathers all states of the diagram for each state-chart connected to a class. Transitions between states are expressed through rules. Each action and event conform to a class operation. Therefore, each action and event is modeled as a dynamic function.

The state-chart diagram models the different states that the "borrowed-Book" class can be in and how that class transitions from state to state. In our example we can borrow a book only if the book is available. To model the loan process is the task of author Author2. So he must use the classes of $C'2$. For this reason he instantiates a contract with $C'2$. See figure 2. The comparable parts of both $C$s are the class-diagram.

The following table shows a part of the coherence matrix for the $C'1$ before the modification.

The following table shows a part of the coherence matrix for $C'2$ before the modification.

$C'1$ and $C'2$ are token identical as they share the same general coherence matrix $GCM$. $C'1$ and $C'2$ are consistent as they have the same value, the values of their general coherence matrix $GCM$ are consistent.

The collaboration contract $C\infty$ is based on $C'1$, $C'2$, and the general coherence matrix $GCM$. These contract defines what each author of a $C'1$, $C'2$ has to do.
exampleCollaborationContract =
if event = generate then
    name(ct) := Collaboration
    agent(ct) :=
        {ClassDesigner, StatechartDesigner, DecomposerService}
    norms(ct) := Decompose, DecomposeReply
    agent(ct) := (Decomposer, Decompose)
    type(n1) := Obligation
    name(n1) := Collaboration
    activation(n1) := 'TryToCommitModification(umlde) ∧
                    ∀umldcProprietor(c)
                Input(ASMSpec(Proprietor(c))) ==
                Input(ASMSpec(Proprietor(c)))
                Output(ASMSpec(Proprietor(c)))! =
                output(ASMSpec(Proprietor(c)))'
    expression(n1) := out(c) :='Permission formodificationallowed?';
    finalization(n1) := 'Allowed(c) ==
                        T ∧ State(c, ObligationDecompoese) == activate'

C1 specifies the following collaboration contract:

Permission:

When any UML diagram cluster involved in the contract will modify the "common" part then it can do all modifications which does not change the Input-Output-Behavior of any associated ASM specification.
\[
\text{ObligationCollaboration}(c) = \begin{array}{l}
\text{begin} \\
\text{if } \text{event}(c) == \text{Collaborate} \text{ then} \\
\quad \text{// Activation} \\
\quad \quad \text{if } \text{TryToCommitModification}(\text{umldc}) \land \forall \text{umldc} \in \text{Proprietor}(c) \\
\quad \quad \quad \text{Input}(\text{ASM Spec}(\text{Proprietor}(c))) == \\
\quad \quad \quad \quad \text{Input}(\text{ASM Spec}(\text{Proprietor}(c))) \\
\quad \quad \quad \text{Output}(\text{ASM Spec}(\text{Proprietor}(c)))! = \\
\quad \quad \quad \quad \text{output}(\text{ASM Spec}(\text{Proprietor}(c))) \text{ then} \\
\quad \quad \quad \quad \quad \text{// Execution} \\
\quad \quad \quad \quad \quad \quad \text{out}(c) := \text{’Permission for modification allowed?’}; \\
\quad \quad \quad \quad \quad \quad \text{State}(c, \text{ObligationDecompose}) := \text{activated} \\
\quad \quad \text{// Finalization} \\
\quad \quad \quad \text{if } \text{Allowed}(c) == \top \land \text{State}(c, \text{ObligationDecompose}) == \text{activated} \text{ then} \\
\quad \quad \quad \quad \text{CommitModification}(\text{Proprietor}(c)); \\
\quad \quad \quad \quad \text{State}(c, \text{ObligationCollaboration}) := \text{completed} \\
\text{end}
\end{array}
\]

\text{Obligation:}

When any UML diagram cluster involved in the contract will modify the
"common" part which modifies the input-output-behavior of any
associated ASM specification, then it has to ask this affected UML
diagram cluster. When all affected UML diagram clusters agree then
the modification can be carried out. The affected UML diagram clus-
ters have to lift the modification in their own cluster. (For instance
the pairwise assignment of UML diagram elements.)

\[
\text{PermissionDecompose}(c) = \begin{array}{l}
\text{begin} \\
\text{if } \text{event}(c) == \text{Decompose} \text{ then} \\
\quad \text{// Activation} \\
\quad \quad \text{if } \text{TryToCommitModification}(\text{umldc}) \land \forall \text{umldc} \in \text{Proprietor}(c) \\
\quad \quad \quad \text{Input}(\text{ASM Spec}(\text{Proprietor}(c))) == \\
\quad \quad \quad \quad \text{Input}(\text{ASM Spec}(\text{Proprietor}(c))) \\
\quad \quad \quad \text{Output}(\text{ASM Spec}(\text{Proprietor}(c))) == \\
\quad \quad \quad \quad \text{output}(\text{ASM Spec}(\text{Proprietor}(c))) \text{ then} \\
\quad \quad \quad \quad \quad \text{// Execution} \\
\quad \quad \quad \quad \quad \quad \text{State}(c, \text{ObligationDecompose}) := \text{activated} \\
\quad \quad \text{// Finalization} \\
\quad \quad \quad \text{if } \text{State}(c, \text{ObligationDecompose}) == \text{activated} \text{ then} \\
\quad \quad \quad \quad \text{CommitModification}(\text{Proprietor}(c)); \\
\quad \quad \quad \quad \text{State}(c, \text{PermissionDecompose}) := \text{completed} \\
\text{end}
\end{array}
\]

\text{Sanction:}
When any UML diagram cluster involved in the contract will commit the modification the "common" part which modifies the input-output-behavior of any associated ASM specification and the affected UML diagram clusters disagree then the affected clusters give this cluster a copy of their own. This cluster can now do anything with this copy.

\[
\text{SanctionDecompose}(c) = \begin{align*}
\text{begin} \\
&\text{if } \text{event}(e) == \text{Decompose}(c) \text{ then} \\
&\quad \text{// Activation} \\
&\quad \quad \text{if } \text{CommitModification}(\text{umlde}) \land \forall \text{umldeProprietor}(c) \\
&\quad \quad \quad \text{Input}(\text{ASMSpec(Proprietor}(c))) == \\
&\quad \quad \quad \text{Input}(\text{ASMSpec(Proprietor}(c))) \land \\
&\quad \quad \quad \text{Output}(\text{ASMSpec(Proprietor}(c)))! = \\
&\quad \quad \quad \text{output}(\text{ASMSpec(Proprietor}(c))) \text{ then} \\
&\quad \quad \quad \quad \text{// Execution} \\
&\quad \quad \quad \quad \quad \text{State}(c,\text{SanctionDecompose}) := \text{activated} \\
&\quad \quad \quad \quad \quad \text{Copy}(\text{umlde}) \\
&\quad \quad \quad \quad \text{// Finalization} \\
&\quad \quad \quad \text{if } (\text{Copy}) \text{ then} \\
&\quad \quad \quad \quad \quad \text{State}(c,\text{SanctionDecompose}) := \text{completed} \\
\text{end}
\end{align*}
\]

During the development time the author of \( \mathcal{C}'2 \) - depicted in Figure 2 - figures out that the loan process for a student is different from that loan process of an employee.

An employee does not have a overdue period, (s)he can borrow the book for an unlimited period. The student must give back the book, when the day of return is reached, else his/her loan is in state overdue. As the author of \( \mathcal{C}'2 \) is interested in decomposing the state-chart diagram he must also change the class diagram depicted Figure 2.

A class diagram needs refinement in order to provide all relevant aspects for a specification. The decomposition approach should also be based on contracts. In Figure ?? objects collaborate to support some higher level behavior. State-charts are related to objects and their states. Therefore, the service handles decomposition of class diagrams and their attached state-chart diagrams.

\( \mathcal{C}'2 \) has a development contract behind the collaboration contract, the former of which is depicted in Figure ??, There is for instance a service defined which helps by semi-automtic decomposition of the class diagram, which is turn depends on the changes of the state-chart diagram. (Copy the attributes and methods from the "borrowABook"-class to the "BorrowedByEmployeeClass" and "BorrowedByStudent"-class.)

The author of \( \mathcal{C}'1 \) decides first to use the driver A for the generation of his cluster. The result of this transformation is presented in Figure ??, But he has also the possibility to use driver B. The result of this transformation is presented in Figure ??, In our example he decides to use Driver A. The contract requires that the driver for a generation must hold constant.
Figure 4. ASM rules for class BorrowedBook and transformation driver B

BorrowedBook = \{1, \ldots , n\}
Method = \{rent, return\}
tupel:
\(\text{BorrowedBook} \times \text{Book} \times \text{PID} \times \text{DueDate} \rightarrow \text{Bool}\)
method\(_m\): \(\text{BorrowedBook} \rightarrow \text{String}\)
MainRule =
for each \(o \in \text{BorrowedBook}\) do
end

Figure 5. ASM rules for class BorrowedBook and transformation driver A

BorrowedBook = \{1, \ldots , n\}
Method = \{borrow, return\}
person : borrowBook \rightarrow \text{int}
method\(_m\): \(\text{BorrowBook} \rightarrow \text{String}\)
MainRule =
for each \(o \in \text{BorrowedBook}\) do
   
   MAINRULE =
   if method\(_a\)(o) == borrowBook then
      
      INITIALIZATION()
   
end
student : BorrowByStudent → int

MainRule =
for each o ∈ borrowByStudent( do
    if method(o) == borrowByStudent( then
        Initialize();
    if method(o) == borrow then
        // Code is generated from the state-chart
        if Actor(o) == Student then
            // precondition
            if ctl\text{state}(o) == undef ∨ ctl\text{state}(o) == isReturned then
                if CheckFunctionalDependency then
                    let o = new (borrowByStudent());
                    borrow(o) = param\_borrow
                    Student(o) = param\_Person
                    Duedate(o) = param\_Duedate
                    borrowByStudent\_FD(param\_borrow, param\_Person) :=
                    param\_Duedate;
                    borrow();
                    // postcondition
                    l2 ctl\text{state}(o) := isBorrowed;
                    ctl\text{stateIsBorrowed}(o) := isNotOD;  

        // Glue function: method(o) == return ⊔ UseCase = return
        if method(o) == return then
            if Actor == Actor\_2 then
                // precondition
                if ctl\text{state}(o) == isBorrowed then
                    return();
                // postcondition
                ctl\text{state}(o) := isReturned
            if Date > DueDate ∧ isStudent(o) == true then
                // precondition
                if ctl\text{state}(o) == isNotOD then
                    ctl\text{stateIsBorrowed}(o) := isOD
        end

Figure 6. Student’s Part of the Generated ASM
Compare the part of the ASM in Figure ?? and the new generated ASM in Figure ???. The input for the ASM in Figure ?? would not work for the ASM in Figure ???. In our case, the needed modification is to alter the input-output behaviour of the bounded ASMSpec of $C'1$. So this author needs the agreement of the author of $C'1$ to modify the class diagram.

![Figure 7. UML Diagram cluster in Version V2](image)

The result is shown in Figure ???. Now, the state-chart diagram of the employee’s loan of books is bound to the BorrowedByEmployee class and the loan of books of the student is bound to the BorrowedByStudent class.

An informal sketch of transformation $t_1$ is shown in Figure 8. This transformation maps each method of a class into a dynamic function. The interface of the method is a monitored dynamic function. The implementation of the method of a class is based on the state-chart diagram. The result of this transformation process is depicted in Figure 8.

$C'2$ is based on the standard class diagram, use case and state-chart type $CT$ and contains the class diagram ”’class2’”, ”’useCase2’” and ”’state-chart2’”. As common part of the contract we define the class Person and BorrowedBook. $C'2$ needs transformation $t_2$, which maps a class diagram in coherence with a state-chart diagram into an ASMSpec2. The contract $c2$ for instance contains the service: Decompose and the same collaboration contract pattern as $C'1$.

First, we specify contracts. The GlueDiagrams service is needed for definition of UML diagram clusters by mappings between several UML diagrams. The result of applying the GlueDiagrams service is an ”’integrated’” ASM specification which is given in Figure 8. The corresponding coupling strategies modify parts are illustrated for the decomposition of the class Person into Student class and Employee class. The ASM specification is affected as well.
ti(driver1) =
For all c element in Classes do
    ClassBuilder(c)

ClassBuilder(class) =
    out(class) := "class " + class;
choose a element in method(class) do
    MethodBuilder(class, a)
    method(class) := method(class) - a

MethodBuilder(class, method) =
    out(class) := out(class) + method + '(' + class + ') " + String
    // Implementation of the method body
    Statechart(class)

Figure 8. ASM based Transformation

We need an ASM transformation function, for example, by the two examples given below. The transformation may be supported by several options. The entry point of the mapping are the attributes of BorrowedBook. The transformation is given by Algorithm 8. Another transformation of the class borrowedBook is given in Figure ??.

In summary, the example shows that contract constraints guard against accidental damage to the ASM specification, by ensuring that authorized changes to the ASM specification do not result in a loss of application consistency. We have given an integration concept based upon contracts for specifying the overlapping part of these ASM views to solve the problem of model integration.

4 Conclusion and Future Work

Typically software is a complex products that has been developed by a team or teams of engineers. Engineering is the creative application to construct or operate the same with full cognizance of their design, to forecast their behaviour under specific operating conditions.

Software engineering has to solve problems as they arise. The obtained solution must satisfy conflicting requirements.

ASM is a formal software specification method that covers software processes and products starting from the abstract specification to the executable implementation. A suitable alliance of UML diagrams and formal methods can give rise to practical software engineering. Our approach is to derive ASM specifications from UML specifications. This allows us to strictly test, validate und verify UML specifications by analyzing ASM specifications.

This paper gives an ASM-based approach which addresses these issues. Our approach supports stepwise refinement, provides a precise semantics for each
step in the development process, and is based on executable semantics. It allows thus early validation of the specification.

Figure ?? compares our approach with the one of UML. UML uses semantic variation points [?] to support intentional degrees of freedom for the 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. The UML approach does not solve neither the consistency problem nor the problems 1 to 4.

UML suffers also from the non-integrated development of different UML diagram types. Currently some of the diagrams such as use cases and activity diagrams are associated by mappings. These mapping are however not refinements [?] in the sense of the ASM approach, for example, changes in activity diagrams do not result in change in uses case diagrams. Often diagrams are not associated at all, see for instance the lacking association between class diagrams and use case diagrams in Figure ??.

The consistency problem among UML diagrams can be seen as the main problem to be tackled here. Our solution has a number of advantages:

**Faithful representation of ASM by various UML diagrams:** The ASM specification is used as the global specification of all diagrams within the UML diagram cluster. The mapping from the UML diagram cluster to an ASM specification allows one to consider all aspects of interest within a UML diagram cluster.

**Faithful coexistence of UML diagrams:** As long as we have been choosing faithful representations of the ASM specification by a number of UML diagrams we may bind any change of one of the UML diagrams by contracts. The main aim of these contracts is to define consistency among these diagrams.

**Contract-based refinement of specifications:** Whenever a refinement is applied to any part of the specification then this refinement is only committed if all contract conditions are satisfied. Otherwise we apply an enforcement methods such as cascading refinement of other specifications, default refinement/modification of other specifications, rejection of the current refinement, or deriving 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. Due to open semantics no solution can be envisioned. We use ASM specifications as the backing specification and demonstrate how UML diagrams can be consistently evolved together.

Software engineering can be seen as the art of writing high-quality programs. ASM semantics might help in achieving this goal. Software engineering consists also of process engineering and project management and of systematic development of phases. We developed an approach to support this as well. The main gain from using ASM in software engineering comes from the ASM foundations for constructing systems. ASM provides a framework and the foundation for developing high-quality systems that are complete, unambiguous, consistent, maintainable, etc.
Topics covered by ASM foundations are

– component architectures of software with explicit views for component diagrams, package diagrams, superstructure metamodel diagrams, and composite structure diagrams;
– rigid description of UML diagrams and their combinations, e.g., use cases, state-charts, interaction diagrams;
– collaboration modeling based on contracts, which specify the exchange frames.

The main achievement of backing software engineering by ASM specifications is however consistency management during the development process. Developer can use the large variety of UML diagrams. The utilization of such a variety of UML diagrams is also required by most industrial partners at the moment and is thus a convincing argument in favor for UML.

We could demonstrate that ASM supports development at different levels of abstraction and provides a basis for information hiding. At the same time, ASM supports verification, validation and model checking. It allows reasoning on program properties and restricting them to local computations. Development backed by ASM provides formal foundations for all specifications and documentation. We have shown in this paper that consistent management of specification with faithful diagrams is not any more a dream but achievable.

Our approach is currently under development. We shall demonstrate the power of this approach by three kinds of diagrams: use case diagrams, state-chart diagrams, and class diagrams. We can claim that the four problems of software engineering mentioned above can be entirely solved for these three kinds of UML diagrams. An additional advantage of our approach of executable ASM is that we can validate, test, or verify systems requirements and specifications.

Acknowledgement. We thank Egon Börger for his comments, discussions, and advice.
Engineering Database Component Ware

Bernhard Thalheim

Christian Albrechts University Kiel, Department of Computer Science, 24098 Kiel, Germany,
email: thalheim@is.informatik.uni-kiel.de

Abstract. Large database applications often have a very complex structuring that complicate maintenance, extension, querying, programming. Due to this complexity systems become unmaintenable. We observe, however, that large database applications often use an implicit structuring into connected components. We propose to initially use this internal structuring for application development. The application architecture is based on database components. Database components can be composed to an application system. This paper shows how components may be developed, composed and applied.

1 Towards Information Systems Engineering

Component-Based Application Engineering.
Software engineering is still based on programming in the small although a number of approaches has been proposed for programming in the large. Programming in the large uses strategies for programming, is based on architectures, and constructs software from components which collaborate, are embedded into each other, or are integrated for formation of new systems. Programming constructs are then pattern or high-level programming units and languages.

The next generation of programming observed nowadays is programming in the world within a collaboration of programmers and systems. It uses advanced scripting languages such as Groovy with dynamic integration of components into other components, standardisation of components with guarantees of service qualities, collaboration of components with communication, coordination and cooperation features, distribution of workload, and virtual communities. Therefore, component engineering will also form the kernel engineering technique for programming in the world. The next generation of software engineering envisioned is currently called as programming by composition or construction. In this case components also form the kernel technology for software and hardware.

Software development is mainly based on stepwise development from scratch. Software reuse has been considered but never reached the maturity for application engineering. Database development is also mainly development in the small. Schemes are developed step by step, extended type by type, and normalized locally type by type. Views are still defined type by type although more complex schemata can be easily defined by extended ER schemata [Tha00].

Therefore, database engineering must still be considered as handicraft work which require the skills of an artisan. Engineering in other disciplines has already gained the maturity for industrial development and application.
Engineering applications have been based on the simple **separation principle**: *Separation of elements which are stable from those elements which are not.* This separation allows *standardization* and *simple integration*. An example is the specification of screws as displayed in Figure 1. Screws have a standardized representation: basic data, data on the material, data on the manufacturing, data on specific properties such as head, etc.

![HERM Representation of the Star Type Screw](image)

**Fig. 1.** HERM Representation of the Star Type Screw

**Complex Applications Result in Large Schemata.** Monographs and database course books usually base explanations on small or ‘toy’ examples. Reality is, however, completely different. Database schemata tend to be large, not surveyable, incomprehensible and partially inconsistent due to application, the database development life cycle and due to the number of team members involved at different time intervals. Thus, consistent management of the database schema might become a nightmare and may lead to legacy problems. The size of the schemata may be very large.

It is a common observation that large database schemata are error-prone, are difficult to maintain and to extend and are not surveyable. Moreover, development of retrieval and operation facilities requires highest professional skills in abstraction, memorization and programming. Such schemata reach sizes of more than 1000 attribute, entity and relationship types. Since they are not comprehensible any change to the schema is performed by extending the schema and thus making it even more complex. Database designers and programmers are not able to capture the schema.

Application schemata could be simpler only to a certain extent if software engineering approaches are applied. The repetition and redundancy in schemata is also caused by

- different usage of similar types of the schema,
- minor and small differences of the types structure in application views, and
- semantic differences of variants of types.

---

1 We use the extended ER model [Tha00] that allows to display subtypes on the basis of unary relationship types and thus simplifies representation.
Therefore, we need approaches which allow to reason on repeating structures inside schemata, on semantic differences and differences in usage of objects. Large schemata also suffer from the deficiency of variation detection: The same or similar content is often repeated in a schema without noticing it.

Techniques to Decrease Complexity in Applications.
Large database schemata can be drastically simplified if techniques of modular modelling such as modular design by units[Tha00] are used. It is an abstraction technique based on principles of hiding and encapsulation. Design by units allows to consider parts of the schema in a separate fashion. The parts are connected via types which function similar to bridges.

Data warehousing and user views are often based on snowflake or star schemata. The intuition behind such schemata is often hidden. Star and snowflake schemata are easier to understand, to query, to survey and to maintain. At the same time, these structures are of high redundancy and restricted modelling power. For instance, the central type in a star or snowflake schema is a relationship type which has attributes that use only numerical types. We may wonder, however, why we need to apply these restrictions and why we should not use this approach in general.

Co-design [Tha00] of database applications aims in consistent development of all facets of database applications: structuring of the database by schema types and static integrity constraints, behavior modelling by specification of functionality and dynamic integrity constraints and interactivity modelling by assigning views to activities of actors in the corresponding dialogue steps. Co-design, thus, is based on the specification of the database schema, functions, views and dialogue steps. At the same time, various abstraction layers are separated such as the conceptual layer, requirements acquisition layer and implementation layer.

Software becomes surveyable, extensible and maintainable if a clear separation of concerns and application parts is applied. In this case, a skeleton of the application structure is developed. This skeleton separates parts or services. Parts are connected through interfaces. Based on this architecture, an application can be developed part by part.

We combine modularity, star structuring, co-design, and architecture development to a novel framework based on components. Such combination seems to be not feasible. We discover, however, that we may integrate all these approaches by using a component-based approach. This skeleton can be refined during evolution of the schema. Then, each component is developed step by step. Structuring in component-based co-design is based on two constructs:

Components: Components are the main building blocks. They are used for structuring of the main data. The association among components is based on ‘connector’ types (called hinge or bridge types) that enable in associating the components in a variable fashion.

Skeleton-based construction: Components are assembled together by application of connector types. These connector types are usually relationship types.

Goals of the Paper.
The paper surveys our approach [Tha02,Tha03a,Tha05] for systematic development of
large database schemata and applies it for database construction based on components and for collaborating component suites. The paper is based on [Fey03,FT02,ST06a,ST04]. We introduce first the concept of database components and then discuss engineering of database applications based on components.

2 Database Components and Construction of Schemes

Database Schemes in a Nutshell.
We use the extended ER model for representation of structuring and behavior generalizing the approach of [PBGG89]. The extended ER model (HERM) [Tha00] has a generic algebra and logic, i.e., the algebra of derivable operations and the fragment of (hierarchical) predicate logic may be derived from the HERM algebra whenever the structure of the database is given.

A database type $\mathcal{S} = (S, O, \Sigma)$ is given by

- a structure $S$ defined by a type expression defined over the set of basic types $B$, a set of labels $L$ and the constructors product (tuple), set and bag, i.e. an expression defined by the recursive type equality
  \[ t = B | t \times \ldots \times t | \{t\} | \{l\} | t : t, \]
- a set of operations defined in the ER algebra and limited to $S$, and
- a set of (static and dynamic) integrity constraints defined in the hierarchical predicate logic with the base predicate $P_S$.

Objects of the database type $\mathcal{S}^C$ are $S$-structured. Classes $\mathcal{S}^C$ are sets of objects for which the set of static integrity constraints is valid.

Operations can be classified into “retrieval” operations enabling in generating values from the class $\mathcal{S}^C$ and “modification” operations allowing to change the objects in the class $\mathcal{S}^C$ if static and dynamic integrity constraints are not invalidated.

A database schema $\mathcal{D} = (\mathcal{S}_1, \ldots, \mathcal{S}_m, \Sigma_G)$ is defined by

- a list of different database types and
- a set of global integrity constraints.

The HERM algebra can be used to define (parameterized) views $\mathcal{V} = (V, O_V)$ on a schema $\mathcal{D}$ via

- an (parameterized) algebraic expression $V$ on $\mathcal{D}$ and
- a set of (parameterized) operations of the HERM algebra applicable to $V$.

The view operations may be classified too into retrieval operations $O^R_V$ and modification operations $O^M_V$. Based on this classification we derive an output view $O^V$ of $\mathcal{V}$ and an input view $I^V$ of $\mathcal{V}$.

In a similar way (but outside the scope of this paper) we may define transactions, interfaces, interactivity, recovery, etc.

Obviously, $I^V$ and $O^V$ are typed based on the type system. Data warehouse design is mainly view design [Tha00].
A database component is a database scheme that has an import and an export interface for connecting it to other components by standardized interface techniques. Components are defined in a data warehouse setting. They consist of input elements, output elements and have a database structuring. Components may be considered as input-output machines that are extended by the set of all states $S^C$ of the database with a set of corresponding input views $I^V$ and a set of corresponding output views $O^V$. Input and output of components is based on channels $K$. The structuring is specified by $S_K$.

The structuring of channels is described by the function $\text{type} : C \rightarrow V$ for the view schemata $V$. Views are used for collaboration of components with the environment via data exchange. In general, the input and output sets may be considered as abstract words from $M^*$ or as words on the database structuring.

A database component $K = (S_K, I^V_K, O^V_K, S^C_K, \Delta_K)$ is specified by

- (static) schema $S_K$ describing the database schema of $K$,
- syntactic interface providing names (structures, functions) with parameters and database structure for $S^C_K$ and $I^V_K, O^V_K$,
- behavior relating the $I^V, O^V$ (view) channels

$$\Delta_K : (S^C_K \times (I^V_K \rightarrow M^*)) \rightarrow \mathcal{P}(S^C_K \times (O^V_K \rightarrow M^*))$$

Components can be associated to each other. The association is restricted to domain-compatible input or output schemata which are free of name conflicts.

Components $K_1 = (S_1, I^V_1, O^V_1, S^C_1, \Delta_1)$ and $K_2 = (S_2, I^V_2, O^V_2, S^C_2, \Delta_2)$ are free of name conflicts if the set of attribute, entity and relationship type names are disjoint.

Channels $C_1$ and $C_2$ of components $K_1 = (S_1, I^V_1, O^V_1, S^C_1, \Delta_1)$ and $K_2 = (S_2, I^V_2, O^V_2, S^C_2, \Delta_2)$ are called domain-compatible if

$$\text{dom}(\text{type}(C_1)) = \text{dom}(\text{type}(C_2)).$$

An output $O^V_1$ of the component $K_1$ is domain-compatible with an input $I^V_2$ of the component $K_2$ if $\text{dom}(\text{type}(O^V_1)) \subseteq \text{dom}(\text{type}(I^V_2))$.

Component operations such as merge, fork, transmission are definable via application of superposition operations [Kud82, Mal70]: Identification of channels, permutation of channels, renaming of channels, introduction of fictitious channels, and parallel composition with feedback displayed in Figure 2.

Fig. 2. The Composition of Database Components

Thus, a component schema is usually characterized by a kernel entity type used for storing basic data, by a number of dimensions that are usually based on subtypes of the entity type which are used for additional properties. These additional properties
are clustered according to their occurrence for the things under consideration. Typically, the component schema uses four dimensions: subtypes, additional characterization, versions and meta-characterizations.

The star schema is the main component schema used for construction.

**A star schema** for a database type $C_0$ is defined by

- the (full) (HERM) schema $S = (C_0, C_1, ..., C_n)$ covering all types on which $C_0$ has been defined,
- the subset of strong types $C_1, ..., C_k$ forming a set of keys $K_1, ..., K_s$ for $C_0$, i.e., $\cup_{i=1}^{s} K_i = \{C_1, ..., C_k\}$ and $K_i \rightarrow C_0$, $C_0 \rightarrow K_i$ for $1 \leq i \leq s$ and $\text{card}(C_0, C_i) = (1, n)$ for $(1 \leq i \leq k)$.
- the extension types $C_{k+1}, ..., C_m$ satisfying the (general) cardinality constraint $\text{card}(C_0, C_j) = (0, 1)$ for $(k + 1 \leq i \leq n)$.

The extension types may form their own $(0, 1)$ specialization tree (hierarchical inclusion dependency set). The cardinality constraints for extension types are partial functional dependencies.

There are various variants for representation of a star schemata:

- Representation based on an entity type with attributes $C_1, ..., C_k$ and $C_{k+1}, ..., C_l$ and specialisations forming a specialization tree $C_{l+1}, ..., C_n$.
- Representation based on a relationship type $C_0$ with components $C_1, ..., C_k$, with attributes $C_{k+1}, ..., C_l$ and specialisations forming a specialization tree $C_{l+1}, ..., C_n$.
  In this case, $C_0$ is a pivot element [BP00] in the schema.
- Representation by be based on a hybrid form combining the two above.

Star schemata may occur in various variants within the same conceptual schema. Therefore, we need variants of the same schema for integration into the schema. We distinguish the following variants:

**Integration and representation variants:** For representation and for integration we can define views on the star type schema with the restriction of invariance of identifiability through one of its keys. Views define ‘context’ conditions for usage of elements of the star schema.

**Versions:** Objects defined on the star schema may be replaced later by objects that display the actual use, e.g., *Documents* are obtained and stored in the *Archive*.

**Variants** replacing the entire type another through renaming or substitution of elements.

**History variants:** Temporality can be explicitly recorded by adding a history dimension, i.e., for recording of instantiation, run, usage at present or in the past, and archiving.

**Lifespan variants** of objects and their properties may be explicitly stored. The lifespan of products in the acquisition process can be based on the *Product-Quote-Request-Response-Requisition-Order-InventoryItem-StoredItem* cycle displayed in Figure 6.

**Meta-Characterization of Components, Units, and Associations.** Utilization information is often only kept in log files. Log files are inappropriate if the utilization or historic information must be kept after the data have been changed.
Database applications are often keeping track of utilization information based on archives. The same observation can be made for schema evolution. We observed that database schemata change already within the first year of database system exploitation. In this case, the schema information must be kept as well.

The skeleton information is kept by a meta-characterization information that allows to keep track on the purpose and the usage of the components, units, and associations. Meta-characterization can be specified on the basis of dockets [SS99] that provide information. The following frames follows the co-design approach [Tha00] with the integrated design of structuring, functionality, interactivity and context. The frame is structured into general information provided by the header, application characterization, the content of the unit and documentation of the implementation.

- on the content (abstracts or summaries),
- on the delivery instruction,
- on the parameters of functions for treatment of the unit (opening without) zooming, breath, size, activation modus for multimedia components etc.
- on the tight association to other units (versions, releases etc.),
- on the meta-information such as resources, restriction, copyright, roles, distribution policy etc.
- on the content providers, content reviewers and review evaluators with quality control policies,
- on applicable workflows and the current status of completion and
- on the log information that enable in tracing the object’s life cycle.

Dockets can be extended to general descriptions of the utilization. The following definition frame is appropriate which classifies meta-information into mandatory, good practice, optional and useful information.

3 Non-Invasive Database Component Composition

Construction Requirements.
Component construction is based on a general component architecture or a skeleton. Each component is developed in separate. The advantage of the strict separation is an increase of modularisation, parameterisability and conformance to standards.

We derive now a none-invasive construction approach which does not change components used for construction. Due to this restriction we gain a number of properties such as adaptivity, seamless gluing, extensibility, aspect separation, scalability, and metamodelling and abstraction.

Components and Harnesses.
The construction is based on harnesses and the application skeleton. The skeleton is a special form of a meta-schema architecture. It consists of a set of components and a set of harnesses for superposition operations. Harnesses are similar to wiring harnesses used in electrotechnics. A harness consists of a set of input-output channels that can be used to combine wrapped components.

Given a sets of components $\mathcal{R} = \{K_1, \ldots, K_m\}$ and labels $\mathcal{L} = \{L_1, \ldots, L_n\}$ with $n \geq m$. Given furthermore a total function $\tau : \mathcal{L} \rightarrow \mathcal{R}$ used for assigning roles to
components in harnesses. The triple \((K, L, \tau)\) is called harness skeleton. The arity of the skeleton is \(n\).

The skeleton is graphically represented by doubly rounded boxes. Components are graphically represented by rounded boxes. The construction may lead to complex components called units.

The example in Figure 3 has been used in one of our projects. Parliamentarians and inhabitants are combined into a component Users. We may use a large variety of positions. A user may use a certain service through some devices. Appointments are based on the usage of services. Tools vary depending on services and on equipment. The final schema contains more than 2,500 attribute, entity, cluster and relationship types. The skeleton of the application is rather simple.

Harvest Filters. Components may be associated in a variety of ways. In the application in Figure 3 the usage of services depends on the properties of parties, the tools they may use, and the services provided. Services, parties, and tools have their own dimensionality. If we use the classical approach to schema development each subtype may cause the introduction of a new usage type. The schema explodes due to the introduction of a large variety of usage type. To overcome this difficulty we introduce filters.

Given component schemata of an \(n\)-ary harness skeleton. A filter of an \(n\)-ary harness is an \(n\)-ary relation defined of the multi-dimensional structure of the components, i.e. on the views defined for the components.

Filters may be represented either graphically or in a tabular form. In our example, we obtain the following filter. Components are already presented in Figure 3. We de-
velop a number of services which might be used depending on the role, rights, and positions of the users. For instance, the parliamentarian is interested in search of related documents in the role of an inhabitant and in search of related meetings.

The implementation of filters is rather straightforward. Each harness has a filter. Since views are defined together with their identification mechanism, an n-ary harness may be represented by an $(n+1)$-ary relationship type associating the components with their roles and extended by the filter.

A harness consists of the harness skeleton $\mathcal{H} = (K, L, \tau)$ and the harness filter $\mathfrak{F} = \{(L_i, V_{L_i}) | 1 \leq i \leq n, L_i \in L, V_{L_i} \subseteq V_{\tau(L_i)}\}$ for a set of wrapped components $(K_i, V_i)$.

Operators Used For Non-Invasive Schema Construction.

In [Tha03b] a number of composition operators for construction of entity and relationship types has been introduced: constructor-based composition, bulk composition, lifespan composition (architecture-based composition, evolution composition, circulation composition, incremental composition, network composition, loop composition), and context composition.

We generalize now these composition operators to component-based schema construction.

Constructor harnesses are based on composition operations such as product, nest, disjoint union, difference and set operators.

Bulk harnesses allow to bound components, types or classes which share the same skeleton. Two harness skeletons $\mathcal{H}_1 = (K_1, L_1 \tau_1)$ and $\mathcal{H}_2 = (K_2, L_2 \tau_2)$ are called
unifiable if they are defined over the same set of components, \(|\Sigma_1| = |\Sigma_2| = n\), and there exists a permutation \(\rho\) on \(\{1, \ldots, n\}\) such that \(K_{\tau_1(i)} = K_{\tau_2(\rho(i))}\). The bulk harness of unifiable harnesses \(H_1, \ldots, H_p\) is constructed by renaming the labels \(L_j\) of each harness \(H_j\) to \(L_{i,j}\) and combining the label functions \(\tau_i\).

**Application-separating harnesses:** An enterprise is usually split into departments or units which run their own applications and use their own data. Sharing of data is provided by specific harnesses.

**Distribution-based harnesses:** Data, functions and control may be distributed. The exchange is provided through specific combinations which might either be based on exchange components that are connected to the sites by harnesses or be based on combination harnesses.

**Application-separation-based harnesses** have been widely used for complex structuring. The architecture of SAP R/3 often has been displayed in the form of a waffle. For this reason, we prefer to call this composition *waffle composition* or *architecture composition* displayed in Figure 4.

![Fig. 4. The Waffle Architecture Composition](image-url)

An Application of Component Composition.

A typical lifespan construction is the *Order* chain displayed in Figure 6. We discover a chain in the ordering and trading process: *Quote, Request, Response, Requisition, Order, Delivery, Billing, Payment*. Within this chain, parameters such as people responsible in certain stages are inherited through the components. They are included into the type for the purpose of simpler maintenance. They cannot be changed within the type inheriting the component. Thus, we use an extended inheritance of structuring beyond the inheritance of identification.

At the same time, this schema can be constructed on the basis of components. We may distinguish only four basic parts. Parties are either organisations or people. Products have a number of properties that are independent on parties. The two components are associated within the ordering and trading process. The parties may play different roles within this process. The parties act based on these roles. So, the component schema is given in Figure 5.

The roles of parties in the ordering and trading process can be unfolded. We observe a role of a supplier, of a requestor, of a responding party, of a requisition party and
finally the role of the orderer. At the same time, the final order has a history or a lifespan. We may apply the lifespan constructor as well. The application can be either based on collaborating components are can be condensed to the schema given in Figure 6. This schema combines components and unfolds roles and expands the ordering and trading activities. We notice that this schema is not necessarily the solution for the ordering and trading process. We may use the components instead and explicitly model component collaboration. In this case the components may stay non-integrated.

4 Collaborating Database Component Suites

Services Provided By Components For Loosely Coupled Suites. A service consists of a wrapped component \((K_i, V_i)\), the competencies \(\Sigma_{(K_i, V_i)}\) provided and properties \(\Psi_{(K_i, V_i)}\) guaranteeing service quality. Wrapped components offer their own data and functions through their views. The competence of a service manifests itself in the set of tasks \(T\) that may be performed and in the guarantees for their quality.
Database Component Collaboration. Instead of expanding and unfolding the component schema in Figure 5 we may follow a different paradigm. The four basic parts are loosely associated by a collaboration, are supported by component databases and communicate for task resolution. This approach has already been tried for distributed databases. Our approach is far more general and provides a satisfying solution.

A collaborating database component suite \( \Theta = (R, H, \xi, \Sigma) \) consists of
- an set \( R \) of wrapped database components \( (K_i, V_i) \)
- a harness consisting of the harness skeleton \( H = (K, L, \tau) \) and the harness filter \( \xi \),
- an collaboration schema \( \xi \) among these components based on the harness, and
- obligations \( \Sigma \) requiring maintenance of the collaboration.

The collaboration schema explicitly models collaboration among components. We distinguish three basic processes of component collaboration:

**Communication** is defined via exchange of messages and information or simply defined via services and protocols [Kö03]. It depends on the choice of media, transmission modes, meta-information, conversation structure and paths, and on the restriction policy. Communication must be based on harnesses.

**Coordination** is specified via management of components, their activities and resources. It rules collaboration. The specification is based on the pre-/post-articulation of tasks and on the description management of tasks, objects, and time. Coordination may be based on loosely or tightly integrated activities, may be enabled, forced, or blocked. Coordination is often specified through contracts and refines coordination policies.

**Cooperation** is the production of work products taking place on a shared space. It can be considered as the workflow or life case perspective. We may use a specification based on storyboard-based interaction that is mapped to (generic and structured) workflows. The information exchange is based on component services [ST06a] for production, manipulation, organization of contributions.

This understanding has become now a folklore model for collaboration but has not yet been defined in an explicit form. We use the separation of concern for the specification of component collaboration.

**Collaboration obligations** are specified through the collaboration style and the collaboration pattern.

The collaboration style is based on four components describing supporting programs of the connected component including collaboration management:

- **Data access pattern** for data release through the net, e.g., broadcast or P2P, for sharing of resources either based on transaction, consensus, and recovery models or based on replication with fault management, and for remote access including scheduling of access;
- the style of collaboration on the basis of component models which restrict possible communication;
- and the coordination workflows describing the interplay among parties, discourse types, name space mappings, and rules for collaboration.
Collaboration pattern generalize protocols and their specification [Kön03]. They include the description of components, their responsibilities, roles and rights. We know a number of collaboration pattern supporting access and configuration (wrapper, facade, component configuration, interceptor, extension interface), event processing (reactor, proactor, asynchronous completion token, accept connector), synchronization (scoped locking, strategized locking, thread-safe interface, double-checked locking optimization) and parallel execution (active object, monitor object, half-sync/half-async, leader/followers, thread-specific storage).

Exchange frames combine the collaboration schema with the collaboration obligations. The collaboration schema can be considered to be an exchange architecture that may also include the workplace of the client using the component suite.

Supporting Collaboration Schemata By Service Managers.
The abstraction layer model [Tha00,ST06b] distinguishes between the application domain description, the requirements prescription, the system specification, and the logical or physical coding. The specification layer typically uses schemata for specification. These schemata may be mapped to logical codings. The mapping of services to logical database components is already given by classical database textbooks. We map collaboration schemata to service managers. This mapping provides also a framework for characterisation of competencies and quality.

The service manager Man supports functionality and quality of services and manages sets of wrapped components. The manager supports a number of features for collaboration. The architecture of the services manager follow the separation of concern into communication, coordination, and cooperation. We may thus envision the architecture in Figure 7.

![Fig. 7. Layers of a services manager for typical collaborating components](image)

Collaborating services are defined by the quadruple $S = (\Sigma, Man, \Psi_S, \Psi_S)$ describing (Collaborating Suite, Service Manager, Competence, Characteristics). The competence is derived from the competence of the services. The quality of collaborating services may also be derived from the quality properties of components in the suite based on the properties of the harnesses, their collaboration schema, and the corresponding obligations. Typically, quality heavily depends on the suite properties. For instance, reliability of a suite may be less than the reliability of its components.
Concluding by Demonstrating the Potential of Privacy Supporting Suites. Let us show the potential of loosely coupled database component suites for privacy workbenches. Privacy research is becoming the “poor cousin” among the mainstream research. Novel applications such as Web2.0 have created a new rush towards social networking and collaborative applications. This enables new possibilities, but also is a threat to users’ privacy and data. On the surface, many people seem to like giving away their data to others in exchange for building communities or like to get bribes from companies in exchange of privacy. A number of hidden privacy implications of some Web2.0 and Identity2.0 services, standards and applications can be observed here. At the same time, it is often stated that there is no way to properly preserve privacy.

We show the potential of collaborating databases based on the infon model of [AFFT05]. An infon is a discrete item of information of an individual and may be parametric. The parameters are objects, and the so-called anchors assign these objects such as agents to parameters.

We may distinguish four relationships between infons and individuals (people), institutions, agencies, or companies: An infon may be possessed by an individual, institution, agency, or company. For example, an individual may possess private information of another individual or, a company may have in its database, private information of someone. Individuals know that an infon is in possession of somebody else. Infons may belong to individuals. Finally, an infon is owned by an individual. The ownership is the basis for the specification of privacy.

The owner sovereignty principle restrains the right or sovereignty of people over their owned infons. A policy supporting the owner sovereignty principle restrains the possessor in the role of ‘content and topic observer’ and preserves the owner in the role of ‘informed owner’ and ‘refresher’. The contract between owner and possessor restricts the possibilities and rights of the possessor for using content and topics on an ongoing basis by additional actions such as

- to monitor activities of the possessor,
- to collect information (about conditions of possession),
- to give a warning to the owner, and
- to take actions such as use, security, welfare, accuracy, correctness, and maintenance of infons to the owner.

The collaboration is faithful if the portfolio and profile of contracting possessor do not include any forbidden action or ability, all reporting obligations are observed, and the proprietor is able to observe obligations applied to the possessor.

The private database is called information wallet if it is a component service with the following additional function enhancements for owners o, possessors p, infons i, infon requests ri, time stamps t, delivered infon streams identifiers si, public keys puk(ri, o, p, t) for p, private keys prik(i, o, p, t) for o, records of delivered infons by the owner store(o, i, p, si), and encoding and decoding relations encrypt(i, prik, si), decrypt(p, ri, si, puk, t) extended by steganographic watermarking mark(i, o, p) for infons:

- \[\text{satisfy}(\text{request}(r_i, o, p, t)) \Rightarrow \text{encrypt}(i, \text{prik}(i, o, p, t), s_i) \land \text{deliver}(p, o, s_i) \land \text{store}(o, i, p, s_i)\]
- \[\text{decrypt}(p, r_i, s_i, puk(s_i, o, p, t), t') \Rightarrow \text{informed}(o, \text{Act}(p, s_i, \text{decrypt}(t)), t') \land \text{mark}(i, o, p)\]
– \( \text{read}(p, \text{mark}(i, o, p), t') \Rightarrow \text{inform}(o, \text{Act}(p, s_i, \text{read}), t') \)
– \( \text{send}(p, \text{mark}(i, o, p), p', t') \Rightarrow \text{inform}(o, \text{Act}(p, s_i, \text{send}(p, p')), t') \wedge \)
– \( \neg\text{send}(p, \text{mark}(i, o, p), p', t') \wedge \text{send}(p, r, p', t') \)
– \( \text{satisfy}(\text{request}(\text{puk}(s_i, o, p, t)), o, p, t') \Rightarrow \)
\( \text{deliver}(p, o, \text{puk}(s_i, o, p, t)) \wedge \text{store}(o, i, p, \text{puk}(s_i, o, p, t)). \)

We assume that watermarked infons cannot be changed by anybody. We can show now that information wallets preserve the owner sovereignty principle.

References


Extended Entity-Relationship Model

Bernhard Thalheim
Christian-Albrechts University Kiel, http://www.informatik.uni-kiel.de/~thalheim

SYNONYMS
EERM, HERM; higher-order entity-relationship model; hierarchical entity-relationship model

DEFINITION
The extended entity-relationship (EER) model is a language for defining the structure (and functionality) of database or information systems. Its structure is developed inductively. Basic attributes are assigned to base data types. Complex attributes can be constructed by applying constructors such as tuple, list or set constructors to attributes that have already been constructed. Entity types conceptualise structuring of things of reality through attributes. Cluster types generalise types or combine types into singleton types. Relationship types associate types that have already been constructed into an association type. The types may be restricted by integrity constraints and by specification of identification of objects defined for a type. Typical integrity constraints of the extended entity-relationship model are participation, look-across, and general cardinality constraints. Entity, cluster, and relationship classes contain a finite set of objects defined on these types. The types of an EER schema are typically depicted by an EER diagram.

HISTORICAL BACKGROUND
The entity-relationship (ER) model was introduced by P.P. Chen in 1976 [1]. The model conceptualises and graphically represents the structure of the relational model. It is currently used as the main conceptual model for database and information system development. Due to its extensive usage a large number of extensions to this model were proposed in the 80's and 90's. Cardinality constraints [1, 3, 4, 8] are the most important generalisation of relational database constraints [7]. These proposals have been evaluated, integrated or explicitly discarded in an intensive research discussion. The semantic foundations proposed in [2, 5, 8] and the various generalisations and extensions of the entity-relationship model have led to the introduction of the higher-order or hierarchical entity-relationship model [8] which integrates most of the extensions and also supports conceptualisation of functionality, distribution [9], and interactivity [6] for information systems. Class diagrams of the UML standard are a special variant of extended entity-relationship models.
The ER conferences (annually; since 1996: International Conference on Conceptual Modeling, http://www.conceptualmodeling.org/) are the main forum for conceptual models and modelling.

SCIENTIFIC FUNDAMENTALS
The extended entity-relationship model is mainly used as a language for conceptualisation of the structure of information systems applications. Conceptualisation of database or information systems aims to represent the logical and physical structure of an information system. It should contain all the information required by the user and required for the efficient behavior of the whole information system for all users. Conceptualisation may further target the specification of database application processes and the user interaction. Structure description are currently the main use of the extended ER model.

An example of an EER diagram.
The EER model uses a formal language for schema definition and diagrams for graphical representation of the
schema. Let us consider a small university application for management of Courses. Proposed courses are based on courses and taught by a docent or an external docent within a certain semester and for a set of programs. Proposals typically include a request for a room and for a time and a categorisation of the kind of the course. Theses proposals are the basis for course planning. Planning may change time, room and kind. Planned courses are held at the university. Rooms may be changed. The example is represented by the EER diagram in Figure 1.

![Figure 1: Extended Entity-Relationship Diagram for Course Management](image)

Entity types are represented graphically by rectangles. Attribute types are associated with the corresponding entity or relationship type. Attributes primarily identifying a type are underlined. Relationship types are represented graphically by diamonds and associated by directed arcs to their components. A cluster type is represented by a diamond, is labelled by the disjoint union sign, and has directed arcs from the diamond to its component types. Alternatively, the disjoint union representation \( \oplus \) is attached to the relationship type that uses the cluster type. In this case directed arcs associate the \( \oplus \) sign with component types. An arc may be annotated with a label.

**The definition scheme for structures.**

The extended entity-relationship model uses a data type system for its attribute types. It allows the construction of entity types \( E \doteq (\text{attr}(E), \Sigma_E) \) where \( E \) is the entity type defined as a pair — the set \( \text{attr}(E) \) of attribute types and the set \( \Sigma_E \) of integrity constraints that apply to \( E \). The definition \( \text{def} \) of a type \( T \) is denoted by \( T \doteq \text{def} \).

The EER model lets users inductively build relationship types \( R \doteq (T_1, \ldots, T_n, \text{attr}(R), \Sigma_R) \) of order \( i \geq 1 \) through a set of (labelled) types of order less than \( i \), a set of attribute types, and a set of integrity constraints that apply to \( R \). The types \( T_1, \ldots, T_n \) are the components of the relationship type. Entity types are of order 0. Relationship types are of order \( i \) if they have only entity types as component types. Relationship types are of order \( i \) if all component types are of order less than \( i \) and if one of the component types is of order \( i - 1 \). Additionally, cluster types \( C \doteq T_1 \cup \ldots \cup T_n \) of order \( i \) can be defined through a disjoint union \( \cup \) of relationship types of order less than \( i \) or of entity types.

Entity/relationship/cluster classes \( T^C \) contain a set of objects of the entity/relationship/cluster type \( T \). The EER model mainly uses set semantics, but (multi-)list or multiset semantics can also be used. Integrity constraints apply to their type and restrict the classes. Only those classes are considered for which the constraints of their types are valid. The notions of a class and of a type are distinguished. Types describe the structure and constraints. Classes contain objects.

The data type system is typically inductively constructed on a base type \( B \) by application of constructors such as the tuple or products constructor \( (..) \), set constructor \( \{..\} \), and the list constructor \( < .. > \). Types may be optional component types and are denoted by \([..]\).

The types \( T \) can be labelled \( l : T \). The label is used as an alias name for the type. Labels denote roles of the type. Labels must be used if the same type is used several times as a component type in the definition of a relationship.
or cluster type. In this case they must be unique.

An entity-relationship schema consists of a set of data, attribute, entity, relationship, and cluster types which types are inductively built on the basis of the base types. Given a base type system \( B \). The types of the ER schema are defined through the type equation:

\[
T = B \mid \langle l_1 : T, ..., l_n : T \rangle \mid \{T\} \mid <T> \mid [T] \mid T \cup T \mid l : T \mid N \equiv T
\]

**Structures in detail.**

The classical four-layered approach is used for inductive specification of database structures. The first layer is the data environment, called the basic data type scheme, which is defined by the system or is the assumed set of available basic data types. The second layer is the schema of a database. The third layer is the database itself representing a state of the application’s data often called micro-data. The fourth layer consists of the macro-data that are generated from the micro-data by application of view queries to the micro-data.

**Attribute types and attribute values.**

The classical ER model uses basic (first normal form) attributes. Complex attributes are inductively constructed by application of type constructors such as the tuple constructor \( (..) \), set constructor \( \{..\} \), and the list constructor \( <..> \). Typical base types are integers, real numbers, strings, and time. Given a set of names \( N \) and a set of base types \( B \), a basic attribute type \( A :: B \) is given by an (attribute) name \( A \in N \) and a base type \( B \). The association between the attribute name and the underlying type is denoted by ::. The base type \( B \) is often called the domain of \( A \), i.e. \( dom(A) = B \). Complex attributes are constructed on base attributes by application of the type constructors. The notion of a domain is extended to complex attributes, i.e. the domain of the complex attribute \( A \) is given by \( dom(A) \). Components of complex attributes may be optional, e.g., the Title in the attribute Name.

Typical examples of complex and basic attributes in Figure 1 are

- Name \( \equiv (\text{FirstNames} <\text{FirstName}>, \text{FamName}, \text{[AcadTitles]}, \text{[FamilyTitle]}) \),
- PersNo \( \equiv \text{EmplNo} \cup \text{SocSecNo} \),
- AcadTitles \( \equiv \{\text{AcadTitle}\} \),
- Contact \( \equiv \langle\text{Phone}(|\text{PhoneAtWork}|, \text{private}), \text{Email}, \text{URL}, \text{WebContact}, [\text{Fax}(|\text{PhoneAtWork}|)]\rangle \),
- PostalAddress \( \equiv \langle\text{Zip}, \text{City}, \text{Street}, \text{HouseNumber}\rangle \)

for DateOfBirth :: date, AcadTitle :: acadTitleType, FamilyTitle :: familyTitleAcronym, Zip :: string7,

- SocSecNo :: string9, EmplNo :: int, City :: varString, Street :: varString, HouseNumber :: smallInt.

The complex attribute Name is structured into a sequence of first names, a family name, an optional complex set-valued attribute for academic titles, and an optional basic attribute for family titles. Academic titles and family titles can be distinguished from each other.

**Entity types and entity classes.**

Entity types are characterized by their attributes and their integrity constraints. Entity types have a subset \( K \) of the set of attributes which serve to identify the objects of the class of the type. This concept is similar to the concept of key known for relational databases. The key is denoted by ID(K). The set of integrity constraints \( \Sigma_E \) consists of the keys and other integrity constraints. Identifying attributes may be underlined instead of having explicit specification.

Formally, an entity type is given by a name \( E \), a set of attributes \( attr(E) \), a subset \( id(E) \) of \( attr(E) \), and a set \( \Sigma_E \) of integrity constraints, i.e.

\[
E \equiv (attr(E), \Sigma_E)
\]

The following types are examples of entity types in Figure 1:

- Person \( \equiv (\{\text{Name}, \text{Login}, \text{URL}, \text{Address}, \text{Contact}, \text{DateOfBirth}, \text{PersNo}\}) \)
- Course \( \equiv (\{\text{CourseID}, \text{Title}, \text{URL}\}, \{\text{ID}(\{\text{CourseID}\})\}) \),
- Room \( \equiv (\{\text{Building}, \text{Number}, \text{Capacity}\}, \{\text{ID}(\{\text{Building}, \text{Number}\})\}) \),
- Semester \( \equiv (\{\text{Term}, \text{Date}(\text{Starts}, \text{Ends})\}, \{\text{ID}(\{\text{Term}\})\}) \).

An ER schema may use the same attribute name with different entity types. For instance, the attribute URL in Figure 1 is used for characterising additional information for the type Person and the type Course. If they need to be distinguished, then complex names such as CourseURL and PersonURL are used.

Objects on type \( E \) are tuples with the components specified by a type. For instance, the object \( (\text{or entity}) (\text{HRS3}, 408A, 15) \) represents data for the Room entity type in Figure 1.
An entity class \( E^C \) of type \( E \) consists of a finite set of objects on type \( E \) for which the set \( \Sigma_E \) of integrity constraints is valid.

Cluster types and cluster classes.
A disjoint union \( \cup \) of types whose identification type is domain compatible is called a cluster. Types are domain compatible if they are subtypes of a common more general type. The union operation is restricted to disjoint unions since identification must be preserved. Otherwise, objects in a cluster class cannot be related to the component classes of the cluster type. Cluster types can be considered as a generalisation of their component types.

A cluster type (or “category”)
\[
C \equiv l_1 : R_1 \cup l_2 : R_2 \cup \ldots \cup l_k : R_k
\]
is the (labelled) disjoint union of types \( R_1, \ldots, R_k \). Labels can be omitted if the types can be distinguished.

The following type is an example of a cluster type:
\[
\text{Teacher} \equiv \text{ExternalDocent} : \text{CollaborationPartner} \cup \text{Docent} : \text{Professor}.
\]
The cluster class \( C^C \) is the ‘disjoint’ union of the sets \( R_1^C, \ldots, R_k^C \). It is defined if \( R_1^C, \ldots, R_k^C \) are disjoint on their identification components. If the sets \( R_1^C, \ldots, R_k^C \) are not disjoint then labels are used for differentiating the objects of clusters. In this case, an object uses a pair representation \((l_i, o_i)\) for objects \( o_i \) from \( R_i^C \).

Relationship types and relationship classes.
First order relationship types are defined as associations between entity types or clusters of entity types. Relationship types can also be defined on the basis of relationship types that are already defined. This construction must be inductive and cannot be cyclic. Therefore, an order is introduced for relationship types. Types can only be defined on the basis of types which have a lower order. For instance, the type \( \text{Professor} \) in Figure 1 is of order 1. The type \( \text{ProposedCourse} \) is of order 2 since all its component types are either entity types or types of order 1. A relationship type of order \( i \) is defined as an association of relationship types of order less than \( i \) or of entity types. It is additionally required that at least one of the component types is of order \( i - 1 \) if \( i > 1 \). Relationship types can also be characterized by attributes. Relationship types with one component type express a subtype or an Is-A relationship type. For instance, the type \( \text{Professor} \) is a subtype of the type \( \text{Person} \).

Component types of a relationship type may be labelled. Label names typically provide an understanding of the role of a component type in the relationship type. Labelling uses the definition scheme \( \text{Label} : \text{Type} \). For instance, the \( \text{Kind} \) entity type is labelled by \( \text{Proposal} \) for the relationship type \( \text{ProposedCourse} \) in in Figure 1. Cluster types have the maximal order of their component types. Relationship types also may have cluster type components. The order of cluster type components of a relationship type of order \( i \) must be less than \( i \).

Component types that are not used for identification within the relationship type can be optional. For example, consider the course planning application in Figure 1. Lectures are courses given by a professor or a collaboration partner within a semester for a number of programs. Proposed courses extend lectures by describing which room is within a semester for a number of programs. Proposed courses extend lectures by describing which room is requested and which time proposals and which restrictions are made. Planing of courses assigns a room to a course that has been proposed and assigns a time frame for scheduling. The kind of the course may be changed. Courses that are held are based on courses planned. The room may be changed for a course. The following types specify these assertions.

\[
\text{ProposedCourse} \equiv (\text{Teacher}, \text{Course}, \text{Proposal} : \text{Kind}, \text{Request} : \text{Room}, \text{Semester}, \text{Set2} : \{\text{Program}\}, \{\text{Time(Proposal, SideCondition)}\}, \Sigma_{\text{ProposedCourse}}).
\]
\[
\text{PlannedCourse} \equiv (\text{ProposedCourse}, [\text{Reassigned} : \text{Kind}], [\text{Reassigned} : \text{Room}], \{\text{TimeFrame, TermCourseID}\}, \Sigma_{\text{PlannedCourse}}).
\]
\[
\text{CourseHeld} \equiv (\text{PlannedCourse}, [\text{Reassigned} : \text{Room}], [\text{StartDate, EndDate, AssistedBy}]), \Sigma_{\text{CourseHeld}}).
\]
The second and third types use optional components in case a proposal or a planning of rooms or kinds is changed. Typically, planned courses are identified by their own term-specific identification. Integrity constraints can be
Formally, a relationship type is given by a name \( R \), a set \( \text{compon}(R) \) of labelled components, a set of attributes \( \text{attr}(R) \), and a set \( \Sigma_R \) of integrity constraints that includes the identification of the relationship type by a subset \( \text{id}(R) \) of \( \text{compon}(R) \cup \text{attr}(R) \), i.e.

\[
R \triangleq (\text{compon}(R), \text{attr}(R), \Sigma_R).
\]

It is often assumed that the identification of relationship types is defined exclusively through their component types. Relationship types that have only one component type are unary types. These relationship types define subtypes. If subtypes need to be explicitly represented then binary relationship types named by \text{IsA} between the subtype and the supertype are used. For instance, the type \text{Professor} in Figure 1 is a subtype of the type \text{Person}. An object (or a “relationship”) on the relationship type \( R \triangleq (R_1, ..., R_n, \{B_1, ..., B_k\}, \text{id}(R), \Sigma_R) \) is an element of the Cartesian product \( R_1 \times \cdots \times R_n \times \text{dom}(B_1) \times \cdots \times \text{dom}(B_k) \). A relationship class \( R^C \) consists of a finite set \( R^C \subseteq R_1 \times \cdots \times R_n \times \text{dom}(B_1) \times \cdots \times \text{dom}(B_k) \) of objects on \( R \) for which \( \text{id}(R) \) is a key of \( R^C \) and which obeys the constraints \( \Sigma_R \).

**Integrity constraints.**

Each database model also uses a set of implicit model-inherent integrity constraints. For instance, relationship types are defined over their component types, and a (relationship) object presumes the existence of corresponding component objects. Typically only finite classes are considered. The EER schema is acyclic. Often names or labels are associated with a minimal semantics that can be derived from the meaning of the words used for names or labels. This minimal semantics allows us to derive synonym, homonym, antonym, troponym, hypernym, and hyponym associations among the constructs used.

The most important class of integrity constraints of the EER model is the class of cardinality constraints. Other classes of importance for the EER model are multivalued dependencies, inclusion and exclusion constraints and existence dependencies[7]. Functional dependencies, keys and referential constraints (or key-based inclusion dependencies) can be expressed through cardinality constraints.

Three main kinds of cardinality constraints are distinguished: participation constraints, look-across constraints, and general cardinality constraints. Given a relationship type \( R \triangleq (\text{compon}(R), \text{attr}(R), \Sigma_R) \), a component \( R' \) of \( R \), the remaining substructure \( R'' = R \setminus R' \) and the remaining substructure \( R''' = R'' \cap_R \text{compon}(R) \) without attributes of \( R \).

The participation constraint \( \text{card}(R, R') = (m,n) \) restricts the number of occurrences of \( R' \) objects in the relationship class \( R^C \) by the lower bound \( m \) and the upper bound \( n \). It holds in a relationship class \( R^C \) if for any object \( o' \in R^C \) there are at least \( m \) and at most \( n \) objects \( o \in R^C \) with \( \pi_{R'}(o) = o' \) for the projection function \( \pi_{R'} \) that projects \( o \) to its \( R' \) components.

 Participation constraints relate objects of relationship classes to objects of their component classes. For instance, the constraint \( \text{card}(\text{ProposedCourse}, \text{SemesterCourse}) = (0,3) \) restricts relationship classes for proposals for courses per semester to at least 0 and at most 3, i.e. each course is proposed at most three times in a semester. There are at most three objects \( o \) in \( \text{ProposedCourse}^C \) with the same course and semester objects. The integrity constraint \( \text{card}(\text{ProposedCourse}, \text{DocentSemester}) = (3,7) \) requires that each docent is giving at least 3 courses and at most 7 courses. External docents may be obliged by other restrictions, e.g., \( \text{card}(\text{ProposedCourse}, \text{ExternalDocentSemester}) = (0,1) \).

Formally, the integrity constraint \( \text{card}(R, R') = (m,n) \) is valid in \( R^C \) if \( m \leq \left| \{ o \in R^C : \pi_{R'}(o) = o' \} \right| \leq n \) for any \( o' \in \pi_{R'}(R^C) \) and the projection \( \pi_{R'}(R^C) \) of \( R^C \) to \( R' \). If \( \text{card}(R, R') = (0,1) \) then \( R' \) forms an identification or a key of \( R \), i.e. \( \text{ID}(R') \) for \( R \). This identification can also be expressed by a functional dependency \( R : R' \rightarrow R'' \).

The lookup or look-across constraint \( \text{look}(R, R') = m..n \) describes how many objects \( o'' \) from \( R'''^C \) may potentially ‘see’ an object \( o' \) from \( R'^C \). It holds in a relationship class \( R^C \) if for any object \( o'' \in \text{dom}(R'''^C) \) there are at least \( m \) and at most \( n \) related objects \( o' \) with \( \pi_{R'}(o) = o' \), i.e. \( m \leq \left| \{ o'' \in \pi_{R'''}(R^C) : o \in R^C \land \pi_{R'}(o) = o' \land \pi_{R'''}(o) = o'' \} \right| \leq n \) for any \( o'' \in \text{dom}(R'''^C) \). Typically, look-across constraints are used for components consisting of one type. Look-across constraints are not defined for relationship types with more than one component type.

Look-across constraints are less intuitive for relationship types with more than 2 component types or with attribute types. For instance, the look-across constraint \( \text{look}(\text{ProposedCourse}, \text{DocentSemester}) = 0..7 \) specifies that for any combination of \text{Teacher}, \text{Room}, \text{Kind}, and \text{Program} objects there are between 0 and 7 \text{Docent} and \text{Semester}
The first constraint does not restrict the database. The second constraint expresses a key or functional dependency.

An EER schema is defined by the pair $\{E_1, \ldots, E_n, C_1, \ldots, C_l, R_1, \ldots, R_m\}$ as components and cluster and relationship types are properly layered.

An EER schema is defined by the pair $D = (S, \Sigma)$ where $S$ is a schema and $\Sigma$ is a set of constraints. A database $D^C$ on $D$ consists of classes for each type in $D$ such that the constraints $\Sigma$ are valid.

The classes of the extended ER model have been defined through sets of objects on the types. In addition to sets, lists, multi-sets or other collections of objects may be used. In this case, the definitions used above can easily be extended.

A number of domain-specific extensions have been introduced to the ER model. One of the most important is the extension of the base types by spatial data types such as: point, line, oriented line, surface, complex surface, oriented surface, line bunch, and surface bunch. These types are supported by a large variety of functions such as: meets, intersects, overlaps, contains, adjacent, planar operations, and a variety of equality predicates.

The translation of the schema to (object-)relational or XML schemata can be based on a profile [8]. Profiles define which translation choice is preferred over other choices, how hierarchies are treated, which redundancy and null-value support must be provided, which kind of constraint enforcement is preferred, which naming conventions are chosen, which alternative for representation of complex attributes is preferred for which types, and whether weak types can be used. The treatment of optional components is also specified through the translation profile.

A profile may require the introduction of identifier types and base the identification on the identifier. Attribute types may be translated into data formats that are supported by the target system.
The EER schema can be used to define views. The generic functions insert, delete, update, projection, union, join, selection and renaming can be defined in a way similarly to the relational model. Additionally, nesting and unnesting functions are used. These functions form the algebra of functions of the schema and are the basis for defining queries. A singleton view is defined by a query that maps the EER schema to new types. Combined views also may be considered which consist of singleton views which together form another EER schema. A view schema is specified over an EER schema $D$ by a schema $V = \{S_1, ..., S_m\}$, an auxiliary schema $A$ and a (complex) query $q : D \times A \rightarrow V$ defined on $D$ and $A$. Given a database $D^C$ and the auxiliary database $A^C$. The view is defined by $q(D^C \times A^C)$.

Graphical representation.
The schema in Figure 1 consists of entity, cluster and relationship types. The style of drawing diagrams is one of many variants that have been considered in the literature. The main difference of representation is the style of drawing unary types. Unary relationship types are often represented by rectangles with rounded corners or by (directed) binary IsA-relationship types which associate by arcs the supertype with the subtype. Tools often do not allow cluster types and relationship types of order higher than 1. In this case, those types can be objectified, i.e. represented by a new (abstract) entity type that is associated through binary relationship types to the components of the original type. In this case, identification of objects of the new type is either inherited from the component types or is provided through a new (surrogate) attribute. The first option results in the introduction of so-called weak types. The direct translation of these weak types to object-relational models must be combined with the introduction of rather complex constraint sets. Typically, this complexity can be avoided if the abstract entity type is mapped together with the new relationship types to a singleton object-relational type. This singleton type is also the result of a direct mapping of the original higher-order relationship type. The diagram can be enhanced by an explicit representation of cardinality and other constraints. If participation constraints $\text{card}(R, R') = (m, n)$ are used for component consisting of one type $R'$ then the arc from $R$ to $R'$ is labelled by $(m, n)$. If look-across constraints $\text{look}(R, R') = m..n$ are used for binary relationship types then the arc from $R$ to $R'$ is labelled by $m..n$.

KEY APPLICATIONS
The main application area for extended ER models is the conceptualisation of database applications. Database schemata can be translated to relational, XML or other schemata based on transformation profiles that incorporate properties of the target systems.

FUTURE DIRECTIONS
The ER model has had a deep impact on the development of diagramming techniques in the past and is still influencing extensions of the unified modelling language UML. UML started with binary relationship types with look-across constraints and without relationship type attributes. Class diagrams currently allow n-ary relationship types with attributes. Relationship types may be layered. Cluster types and unary relationship types allow for distinguishing generalisation from specialisation.

ER models are not supported by native database management systems and are mainly used for modelling of applications at the conceptual or requirements level. ER schemata are translated to logical models such as XML schemata or relational schemata or object-relational schemata. Some of the specifics of the target models are not well supported by ER models and must be added after translating ER schemata to target schemata, e.g., specific type semantics such as list semantics (XML) or as special ordering or aggregation treatment of online analytical processing (OLAP) applications.

The ER model has attracted a lot of research over the last 30 years. Due to novel applications and to evolution of technology old problems and novel problems are challenging the research on this model. Typical old problems that are still not solved in a satisfactory manner are: development of a science of modelling, quality of ER schemata, consistent refinement of schemata, complex constraints, normalisation of ER schemata, normalisation of schemata in the presence of incomplete constraint sets. Novel topics for ER research are for instance: evolving schema architectures, collaboration of databases based on collaboration schemata, layered information systems.
and their structuring, schemata with redundant types, ER schemata for OLAP applications. Structures of database applications are often represented through ER models. Due to the complexity of applications, a large number of extensions have recently been proposed, e.g., temporal data types, spatial data types, OLAP types and stream types. Additionally, database applications must be integrated and cooperate in a consistent form. The harmonisation of extensions and the integration of schemata is therefore a never ending task for database research.

ER models are currently extended for support of (web) content management that is based on structuring of data, on aggregation of data, on extending data by concepts and on annotating data sets for simple reference and usage. These applications require novel modelling facilities and separation of syntactical, semantical and pragmatic issues. The ER model can be extended to cope with these applications. The ER model is mainly used for conceptual specification of database structuring. It can be enhanced by operations and a query algebra. Operations and the queries can also be displayed in a graphical form, e.g., on the basis of VisualSQL. Most tools supporting ER models do not currently use this option. Enhancement of ER models by functionality is necessary if the conceptualisation is used for database development. Based on functionality enhancement, view management facilities can easily be incorporated into these tools. ER models are becoming a basis for workflow systems data. The standards that have been developed for the specification of workflows have not yet been integrated into sophisticated data and application management tools.

URL TO CODE
http://www.informatik.uni-kiel.de/~thalheim/HERM.htm
http://www.is.informatik.uni-kiel.de/~thalheim/indereerm.htm

Readings on the RADD project (Rapid Application and Database Development)

CROSS REFERENCE
I. DATABASE FUNDAMENTALS
   a. Data models (including semantic data models)
   b. Entity-Relationship (ER) model
   c. Unified modelling language (UML)
III. THEORETICAL ASPECTS
   b. Relational Theory

RECOMMENDED READING
Between 3 and 15 citations to important literature, e.g., in journals, conference proceedings, and websites.

Specialisation and Generalisation

Bernhard Thalheim

Christian-Albrechts University Kiel, http://www.informatik.uni-kiel.de/~thalheim/HERM.htm

SYNONYMS
refinement, abstraction, hierarchies;
clustering, grouping, inheritance

DEFINITION
Generalisation and specialisation are main principles of database modelling. Generalisation maps or groups types or classes to more abstract or combined ones. It is used to combine common features, attributes, or methods. Specialisation is based on a refinement of types or classes to more specific ones. It allows developers to avoid null values and to hide details from non-authorised users. Typically, generalisations and specialisations form a hierarchy of types and classes. The more specialised classes may inherit attributes and methods from more general ones. In database modelling and implementation clusters of types to a type that represents common properties and abstractions from a type are the main kinds of generalisations. Is-A associations that specialise a type to a more specific one and Is-A-Role-Of associations that considers a specific behaviour of objects are the main kinds of specialisations.

MAIN TEXT
Specialisation introduces a new entity type by adding specific properties belonging to that type which are different from the general properties of its more general type. Generalisation introduces the Role-Of relationship or the Is-A relationship between a subtype and its general type. Therefore, the application, implementation, and processes are different. For generalisation the general type must be the union of its subtypes. The subtypes can be virtually clustered by the general type. This tends not to be the case for specialisation. Specialisation is a refinement or restriction of a type to more special ones. Typical specialisations are Is-A and Has-Role associations. Exceptions can be modelled by specialisations.

Different kinds of specialisation may be distinguished: structural specialisation which extends the structure, semantic specialisation which strengthens type restrictions, pragmatical specialisation which allows to separate the different usage of objects in contexts, operational specialisation which introduces additional operations, and hybrid specialisations. Is-A specialisation requires structural and strong semantic specialisation. Is-A-Role-Of specialisation requires structural, pragmatical and strong semantic specialisation.

Generalisation is based either on abstraction or on grouping. The cluster construct of the extended ER model is used to represent generalisations. Generalisation tends to be an abstraction in which a more general type is defined by extracting common properties of one or more types while suppressing the differences between them. These types are subtypes of the generic type. New types are created by generalizing classes that already exist. Structural combination typically assumes the existence of a unifiable identification of all types. Semantical combination allows the disjunction of types through the linear sum of semantics. Pragmatical generalisation is based on building collections whenever applications require a consideration of commonalities.

CROSS REFERENCE
I. DATABASE FUNDAMENTALS
   a. Data models (including semantic data models)

REFERENCES
Abstraction

Bernhard Thalheim

Christian-Albrechts University Kiel, http://www.informatik.uni-kiel.de/~thalheim/HERM.htm

SYNONYMS
component abstraction, localisation abstraction, implementation abstraction;
association, aggregation, composition, grouping, specialisation, generalisation, classification

DEFINITION
Abstraction allows developers to concentrate on the essential, relevant or important parts of an application. It uses a mapping to a model from things in reality or from virtual things. The model has the truncation property, i.e. it lacks some of the details in the original, and a pragmatic property, i.e. the model use is only justified for particular model users, tools of investigation, and periods of time. Database engineering uses construction abstraction, context abstraction and refinement abstraction. Construction abstraction is based on the principles of hierarchical structuring, constructor composition, and generalisation. Context abstraction assumes that the surroundings of a concept are commonly assumed by a community or within a culture and focuses on the concept, turning away attention from its surroundings such as the environment and setting. Refinement abstraction uses the principle of modularisation and information hiding. Developers typically use conceptual models or languages for representing and conceptualising abstractions. The enhanced entity-relationship model schema are typically depicted by an EER diagram.

MAIN TEXT
Database engineering distinguishes three kinds of abstraction: construction abstraction, context abstraction and refinement abstraction.
Constructor composition depends on the constructors as originally introduced by J. M. Smith and D.C.W. Smith. Composition constructors must be well founded and their semantics must be derivable by inductive construction. There are three main methods for construction: development of ordered structures on the basis of hierarchies, construction by combination or association, and construction by classification into groups or collections. The set constructors $\subset$ (subset), $\times$ (product) and $\mathcal{P}$ (powerset) for subset, product and nesting are complete for the construction of sets.
Subset constructors support hierarchies of object sets in which one set of objects is a subset of some other set of objects. Subset hierarchies are usually a rooted tree. Product constructors support associations between object sets. The schema is decomposed into object sets related to each other by association or relationship types. Power set constructors support a classification of object sets into clusters or groups of sets - typically according to their properties.

Context abstraction allows developers to commonly concentrate on those parts of an application that are essential for some viewpoints during development and deployment of systems. Typical kinds of context abstraction are component abstraction, separation of concern, interaction abstraction, summarisation, scoping, and focusing on typical application cases.
Component abstraction factors out repeating, shared or local patterns of components or functions from individual concepts. It allows developers to concentrate on structural or behavioral aspects of similar elements of components. Separation of concern allows developers to concentrate on those concepts that are a matter of development and to neglect all other concepts that are stable or not under consideration. Interaction abstraction allows developers to concentrate on those parts of the model that are essential for interaction with other systems or users. Summarisation maps the conceptualisations within the scope to more abstract concepts. Scoping is typically used to select those concepts that are necessary for current development and removes those concepts that do not have an impact on the necessary concepts.
Database models may cover a large variety of different application cases. Some of them reflect exceptional,
abnormal, infrequent and untypical application situations. Focusing on typical application cases explicitly separates models for the normal or typical application case from those that are atypical. Atypical application cases are not neglected but can be folded into the model whenever atypical situations are considered.

The context abstraction concept is the main concept behind federated databases. Context of databases can be characterized by schemata, version, time, and security requirements. Sub-schemata, types of the schemata or views on the schemata, are associated by explicit import/export bindings based on a name space. Parametrisation lets developers to consider collections of objects. Objects are identifiable under certain assumptions and completely identifiable after instantiation of all parameters.

Interaction abstraction allows developers to display the same set of objects in different forms. The view concept supports this visibility concept. Data is abstracted and displayed in various levels of granularity. Summarisation abstraction allows developers to abstract from details that are irrelevant at a certain step. Scope abstraction allows developers to concentrate on a number of aspects. Names or aliases can be multiply used with varying structure, functionality and semantics.

Refinement abstraction is mainly about implementation and modularisation. It allows developers to selectively retain information about structures. Refinement abstraction is defined on the basis of the development cycle (refinement of implementations). It refines, summarises and views conceptualizations, hides or encapsulates details or manages collections of versions. Each refinement step transforms a schema to a schema of finer granularity. Refinement abstraction may be modelled by refinement theory and infomorphisms.

Encapsulation removes internal aspects and concentrates on interface components. Blackbox or graybox approaches hide all aspects of the objects under consideration. Partial visibility may be supported by modularisation concepts. Hiding supports differentiation of concepts into public, private (with the possibility to be visible to ‘friends’) and protected (with visibility to subconcepts). It is possible to define a number of visibility conceptualizations based in inflection. Inflection is used for the injection of combinable views into the given view, for tailoring, ordering and restructuring of views, and for enhancement of views by database functionality. Behavioral transparency is supported by the glassbox approach. Security views are based on hiding. Versioning allows developers to manage a number of concepts which can be considered to be versions of each other.

CROSS REFERENCE
I. DATABASE FUNDAMENTALS
   a. Entity-Relationship Model, Extended Entity-Relationship Model, Object Data Models, Object Role Modeling, Unified Modeling Language

REFERENCES
