Conceptual Content Management for Software Engineering Processes

Sebastian Bossung, Hans-Werner Sehring, Michael Skusa, and Joachim W. Schmidt

{sebastian.bossung,hw.sehring,skusa,j.w.schmidt}@tu-harburg.de
Software Technology and Systems Institute (STS)
Hamburg University of Science and Technology (TUHH)

Abstract. A major application area of information systems technology and multimedia content management is that of support systems for engineering processes. This includes the particularly important area of software engineering. Effective support of software engineering processes requires large amounts of content (texts, diagrams, code, data, executables etc.) from different conceptual domains. The term "software crisis" disappeared gradually when content modelling and management addressed domains from application analysis and system design in addition to the sheer computational code domain.
In this paper we introduce an innovative conceptual content model and apply it in support of software engineering processes and their artefacts. We base our approach on the core model of the computational domain which abstracts computational content (bodies of function code) by the computational concept of signatures (lists of typed function parameters). We generalise this *functional abstraction* model beyond the computational domain by introducing the notion of *asset abstraction* which models entities domain-independently by general content-concept pairs. We introduce an asset language and discuss the essentials of an asset system implementation.
In the application part of the paper we argue that software engineering can be substantially simplified by modelling SE entities from all the domains involved in an SE process homogeneously in an asset-oriented approach—entities ranging from application domains over intermediate architectural and design domains down to the computational domain. Furthermore, we discuss how the mappings between such domains can be substantially supported by services based on asset-oriented information systems.

1 Introduction: Content Management for Software Engineering

Heavy demands for data modelling and content management support dominate all kinds of engineering processes and are the major reasons for the ongoing commercial and scientific success of database technology. A wide variety of domain-specific data and content models has been developed and applied to computer-aided engineering environments.

An engineering area of specific interest and challenge to computer scientists is their home ground of software engineering (SE) (e.g., [26]). SE processes are particularly demanding since they span a wide variety of domains ranging from application entities in the analysis phase via intermediate entities required for system design and architecture down to computational entities for software implementation and execution.

The content models involved in SE processes usually include separate models for texts, diagrams, code, data, executables etc. This heterogeneity of the models causes much of the complexity of SE processes. SE environments, instead of providing homogeneous working support, are often subdivided into disparate tool contexts for diverse domains and their preferred representations: texts for analysis, diagrams for designs, code for executables, etc. This subdivision puts severe limitations particularly on those process steps which have to span several SE phases over various domains such as mapping steps, coherence tests or simple search and navigation tasks. This situation is only partially improved by approaches like those presented, e.g., in [4, 27].

Consequently, we see a demand for a conceptual content model which can be homogeneously applied to all the domains involved in SE processes: to entities from application domains, to those in the intermediate architectural and design area as well as to entities from the computational domain.

In this paper we introduce an innovative conceptual content model applicable to a wide variety of domains and we apply that model to SE processes and their artefacts.

In section 2 we base our content model on the core model of the computational domain which abstracts computational content by the computational concept of signatures. This *functional abstraction* model is then generalised beyond the computational domain (section 3) by introducing the notion of *asset abstraction* which models entities domain-independently. We introduce an asset language and discuss the essentials of asset-based information system implementation. In the application part of the paper (section 4) we discuss how SE can be substantially simplified by modelling all SE entities homogeneously in an asset-oriented model. Furthermore, we argue that the mappings between such domains and other domain-spanning tasks can be supported by services based on asset-oriented information systems. The paper concludes with a short summary and a task outlook in section 5.

2 Conceptual Modelling of Computational Entities

It is historically interesting to observe, how means of abstraction available in programming languages evolved over time. In the assembly language of the early days, the main abstraction was providing human-understandable names (mnemonics) for operation codes. Later more and more abstraction mechanisms were introduced, amongst them functional abstraction and typing (see, e.g., [14] on the history of programming languages).

2.1 Functional Abstraction and Types

Most of even the smallest programs will exhibit near duplicate code if written in a sequential fashion. Introducing the concept of functions to parameterise and factor out this code has several benefits: (1) It makes understanding the program easier as it is broken down into semantically self-contained pieces, (2) it facilitates maintenance, as bugs only have to be fixed in a single place, and (3) it reduces the program's size. Functions usually consist of two important parts: the signature (its formal parameters and its return type) and the function body (the implementing code) [5].

The main power of functions thus lies in the introduction of an abstraction layer that associates conceptual information (the signature) with the code body of the function and hides implementational details. Right from the early days functions had a strong formal foundation to build on: the λ-calculus [16], which provides the theoretical basis for general function semantics: function definition (abstraction) and function invocation (application) form central parts.

Essentially any program deals with data, which the computer handles in the form of a particular internal representation. Generally speaking, any computer-representable data can be see as a string of bits. Depending on its proper interpretation, different operations are possible, for example addition, concatenation, or execution. To support the programmer in making the appropriate assumptions on the interpretation of data, programming languages introduced the concept of types. Types have become a central part of most computer languages by allowing the definition of appropriate computational entities and by formally checking the correctness of their use.

2.2 On Function Signatures over Function Code

Conceptual information on a function is captured in the function's signature to provide enough information to anybody who wants to use the function. The evolution of programming languages brought along new features that take the mechanism of signatures to higher levels of abstraction. Signatures can be found on function, class, and even component level, though the latter is still subject to research [3].

Computational entities are usually modelled in a dualistic way as pairs of code and signature, or, more generally speaking, of value and type. In section 3 we generalise this model to non-computational domains by means of content-concept pairs.

2.3 Operational Support for Computational Model Coherence

Based on functional abstraction, coherence of collections of computational entities can be supported by various technologies including compilers, linkers, runtime bindings, and remote invocations across system boundaries. Compilers make use of type abstractions as well as function signatures in, e.g., decorated abstract syntax trees or symbol tables [1]. This enables important features of compiler technology such as type checking, late binding or type coercion. In addition, by means of function signatures, functions themselves become first class citizens. This allows the introduction of higher-order functions [10].

Function signatures and typing enable runtime systems to select and bind computational entities. A common application is polymorphism.

A third field of application is that of cross system communication. Here signatures are of primary importance, as the full implementation is usually not available to a remote caller. Instead, calling programs are written against signatures, which serve as a language of mutual understanding to both systems. In the same context, named types allow the (un)marshalling of data to be communicated between the systems.

3 Conceptual Modelling of General Domain Entities

The concept of signatures (section 3.1) can be generalised to represent entities of any domain (section 3.2). We introduce a conceptual content modelling language (section 3.3) which drives the automatic generation of *conceptual content management system* (*CCMS*) (section 3.4).

3.1 Functional Abstraction as Special Case of Entity Description

Insights into type systems and their achievements influenced the definition, implementation and utilisation of a series of languages [18, 19, 10]. Starting from Pascal/R and DBPL this led to the orthogonally persistent object systems Tycoon-1 and 2 (Typed Communicating Objects in Open Environments 2) [7].

Viewing functional abstraction from a content management perspective, the code of a function body can be seen as well-formed content which is abstracted by a signature for management purposes such as type checking, late binding, information hiding etc. (see section 2.3).

In a series of application projects we applied the general idea—viewing code as computational content and describing it by a signature as its computational concept—to entity descriptions for arbitrary domains.

Often content is used to describe real-world entities—concrete or abstract ones. Just like a piece of code can be used properly only if its signature is known to the caller, content descriptions of the actual entities have to be paired with a conceptual understanding of the entities' nature. E.g., (John, Smith, 5000) represents the customer John Smith whose balance is $5000 only if the conceptual model [2] of a customer is clear to the viewer.

Therefore, entity descriptions in general consist of content coupled with a conceptual model of the kind of entity it refers to. For such *[content, concept]* pairs we use the notion of an *asset* as an indivisible union of perceivable content and a set of expressions describing it abstractly. This notion is detailed in the subsequent sections.

Managing entities from the computational domain can be understood as a special case of general entity descriptions. Returning to the example of function code, one can view source code as a specific kind of text which follows the constraints of a certain programming language. Therefore, the existence of a pair *[text, Java program]* augments a text to Java source code.

3.2 Assets: On Concept-Content-oriented Modelling

The notion of an asset as introduced in the previous section has been developed in projects carried out in cooperation with project partners from the humanities. One main source of insights is the project *Warburg Electronic Library* (*WEL*) [21]. In this project we support art historians from the domain of political iconography.

For general domain models much can be reclaimed from computational models: the pairing of content and concept, the subsumption of content of specific concepts under more general concepts, the substitutability of content from sub concepts of a given concept etc.

Nevertheless, in some respects the entity models we looked at are fundamentally different from computational models. There is a duality of structure and domain semantics of assets. In the above example records with a structure (first name, family name, balance) can describe domain entities of both the kinds "debtor" as well as "creditor", e.g., depending on the balance.

The most severe distinction between descriptions of computational entities and entities in general is the subjectivity of the latter. For computational domains there exists exactly one well-defined conception. However, in "soft sciences" like the humanities there is no agreed-upon interpretation of contents and thus no single asset class for entity descriptions. Not only does the conceptual modelling of entities evolve over time as new findings lead to a better understanding of a domain, interpretations furthermore coexist as personalised views on entities.

Besides personalisation, there is an additional reason for coexisting asset models. Typically, domains are defined by using assets from existing base domains, which allows for reuse and also leaves asset definition to the experts of the field. To be able to incorporate base domain models into (multidimensional) derived domain models, inter-model relationships need to be established.

For both these reasons—subjective views and reuse—there is a demand for *openness* and *dynamics*. We call a CCMS open if it allows users to define assets according to their current information needs. Assets may change with time or context of the user and can be adapted to personal views. Dynamics is the ability of a system to follow redefinitions of assets at runtime without interrupting the users' work.

Just as content needs to be paired with a concept, open and dynamic systems cannot be based on a data model alone. Data models are limited by technical constraints of the target system (a database in most cases). To avoid such technical aspects in the domain model, a conceptual model is required. We briefly introduce our *asset language* for specifying such a model.

3.3 The Asset Model and Asset Language

In this section we give a brief description of the asset language as far as it is required for this paper. More details on the language can be found in [20, 23, 22].

A *model* consists of asset class definitions for entity descriptions. We refer to the corresponding part of the language as the *asset definition language*. First of all, it (intensionally) describes the structure of assets.

As an example, consider the following asset class definition:

```
class RegentImage {
   content image : Image
   concept characteristic title  : String
           characteristic epoch  : Epoch
           relationship   regent : Regent
           relationship   artist : Artist
           constraint     epoch = artist.epoch
}
```

In the `content` compartment a list of handles for multimedia content objects is given. Possible handle types are determined by a base language which is embedded in the asset language. Currently, we use Java as such a base language.

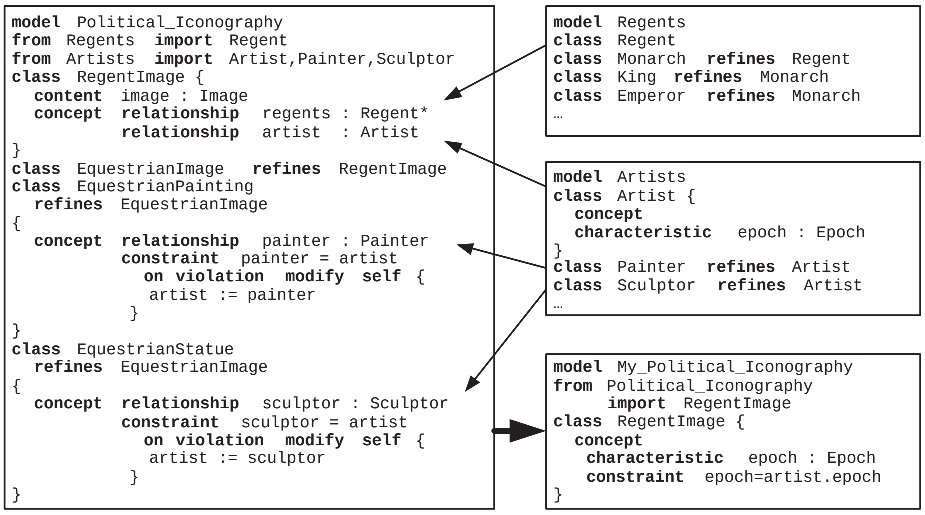
The `concept` compartment consists of a set of conceptual attributes and expressions. Characteristic attributes are ones that are inherent in an entity. In the above example, every `RegentImage` has a `title` and an `epoch` in which it was created. Just as for the content handles, possible values of characteristics are determined by the base language. Relationships are established between assets which describe autonomous entities. Here, each `RegentImage` has references to the depicted `Regent` and to the `Artist` who created that image. Constraints are imposed on assets of a class. In the above example it is required that the `epoch` in which a `RegentImage` has been created is the same as that of the associated `artist`.

While asset classes capture the structural aspects of assets, they can also be defined extensionally by naming a set of asset instances:

class DeathOfTheRegent **definedby** a_1, ..., a_n

Asset definitions are organised in models under the keyword `model`. As an example for the incorporation of base models (see the previous section) consider the sample models shown in fig. 1. One base model called `Regents` defines asset classes for descriptions of regents like kings or emperors. Another base model, `Artists`, likewise defines various classes of artists. Using these two domains as base domains, a new third domain on political iconography can be defined. It incorporates class definitions from the base models.

As can be seen in the example regent and artist information is reused in the political iconography. From the iconography point of view regent and artist information are objective so that one concrete model each is selected and used. Users from the field of political iconography build on these (objective) research findings in their (subjective) entity descriptions.



```
model Political_Iconography
from Regents import Regent
from Artists import Artist,Painter,Sculptor
class RegentImage {
  content image : Image
  concept relationship regents : Regent*
          relationship artist  : Artist
}
class EquestrianImage  refines RegentImage
class EquestrianPainting
  refines EquestrianImage
{
  concept relationship painter : Painter
          constraint painter = artist
            on violation modify self {
             artist := painter
            }
}
class EquestrianStatue
  refines EquestrianImage
{
  concept relationship sculptor : Sculptor
          constraint sculptor = artist
            on violation modify self {
             artist := sculptor
            }
}
```

```
model Regents
class Regent
class Monarch refines Regent
class King refines Monarch
class Emperor refines Monarch
…
```

```
model Artists
class Artist {
  concept
  characteristic  epoch : Epoch
}
class Painter refines Artist
class Sculptor refines Artist
…
```

```
model My_Political_Iconography
from Political_Iconography
     import RegentImage
class RegentImage {
  concept
   characteristic  epoch : Epoch
   constraint epoch=artist.epoch
}
```

Fig. 1. Example of a model composed from base models

Subjectivity is possible because the openness property of the asset language allows the redefinition of assets. As an example, a user can change `RegentImage` by the definition shown in the model `My_Political_Iconography` in fig. 1. In the example a user added an additional characteristic attribute `epoch` plus a constraint. All content handles and conceptual definitions which are not named remain unchanged in the redefined class.

3.4 Conceptual Content Management System Implementation

Openness and dynamics as required for entity descriptions are not covered by contemporary information systems (ISs). Since ISs are usually based on database technology they share its typical constraints, the most crucial being that databases rely on one static schema.

Our approach to open and dynamic CCMSs is based on our asset definition language (see previous section). From models given in the asset definition language—by end-users—with little regard to implementation constraints open dynamic systems are generated by a technology that resembles model-driven architecture approaches [12]. It consists of a *model compiler* and a *modularised architecture* for CCMSs.

A system consists of a set of *components* reflecting one model each. These are broken down into *modules*. The model compiler creates modules, which are the basis of a domain-specific software architecture suitable for dynamic system generation [28]. The functionality of a component is defined by a component *configuration*.

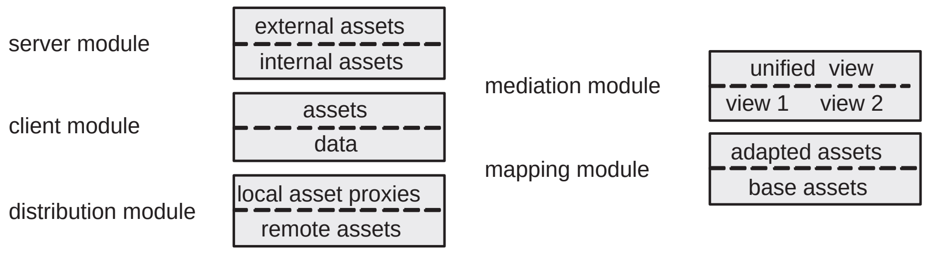


Fig. 2. Modules interface with each other in a layered architecture

Substitutability of modules is achieved by a separation of concerns. For our current purposes we identified five kinds of modules (see fig. 2):

- The description data of an asset (content, characteristics, and relationships) is stored in third party systems, databases in most cases. Mapping asset models to schemata of such systems is done by *client modules*.
- By use of *distribution modules* components can reside at different physical locations and communicate by exchanging data, e.g., XML documents generated from the asset definitions (comparable to the approach of [24]).
- Components are accessed via *server modules* using standard protocols.
- A central building block of the architecture of most CCMS applications is the mediator architecture [29]. In our approach it is implemented by modules of two kinds. One are *mediation modules* which delegate requests to other modules based on the request (operation and assets involved).
- The other kind of modules for the mediator architecture are *mapping modules*. By encapsulating mappings in such modules, rather than integrating this functionality into other modules, mappings can be added dynamically [11].

According to the two ways of combining asset models—model interrelation and personalisation—openness and dynamics in CCMSs happen along two dimensions: (1) the *organisation* and (2) the *application structure* [22]. Along the organisation structure users can define their own views (by personalising content and schema). Along the application structure, entity descriptions are shared and reused across domains.

In our approach the architecture of the generated systems allows changes along the organisation structure by its ability to enable dynamic system evolution through open redefinition of assets and dynamic invocation of the model compiler [23].

The association of models is realised by component configurations. Following the example from the previous section fig. 3 shows a configuration which combines two domains—regent and artists descriptions—into the new domain of political iconography. The component is accessed via mediation module m_{med1}. It distributes requests according to the type of the assets on which operations

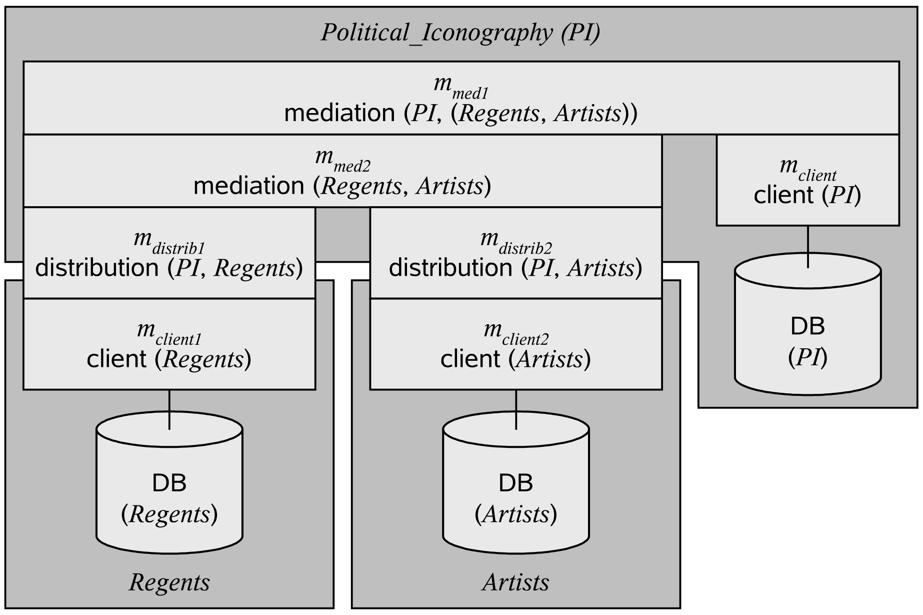


Fig. 3. Sample configuration of a system for a derived model

are invoked. If assets from one of the base domains *Regents* or *Artists* are affected, requests are delegated to the mediation module m_{med2}. This mediation module similarly delegates requests further to one of the components holding theses models. These components are accessed via distribution modules $m_{distrib1}$ and $m_{distrib2}$. In the example of fig. 3 the components consist of client modules $m_{client1}$ and $m_{client2}$ and the respective base system only. Requests to the derived model *Political_Iconography* are forwarded by m_{med1} to the client module m_{client} which manages the users' assets from the political iconography.

As can be seen in fig. 3 the components for *Regents* and *Artists* are integrated into the overall CCMS without modification. This way the cooperating components remain unaffected, thus preserving their autonomy.

4 Asset Modelling and Software Engineering

The engineering processes of non-trivial software lead to a vast number of interrelated, but not explicitly connected, artefacts (e.g., requirements texts, various diagrams, code, tests, executables). In software development methodologies that are common practice today, most of the relations between artefacts of different type are not explicitly modelled. Instead, they are captured in the general knowledge of the developers or by "obvious" choice of naming. Both approaches lead to difficulties: The general knowledge of developers tends to diminish over time and obvious naming is usually only obvious to the one who chose it [13]. Thus, tool support for explicit modelling of such inter-dependencies is highly desirable. In fact, these interrelations are right at the heart of software engineering, as the transitions between development phases happen along them [9].

We therefore propose to model software artefacts by a domain independent conceptual content model, which is based on the asset technology discussed in section 3. This supports (1) retrieval of artefacts, (2) enforcement of their coherence, (3) a common and concise representation, and (4) exchange due to built-in interoperability.

4.1 On Content Linking and Selection

The asset modelling of computational entities aims to integrate content representations across formats and standards. Due to the common conceptual model a CCMS can work with all entities alike, regardless of the tool that supports this particular content format. Note that assets therefore take a completely unintrusive approach to the content that allows for complete owner autonomy. Developers can continue to use the traditional tools for creating and modifying the respective artefacts.

Still, it is possible to guarantee the consistency of changes to system artefacts. Such consistency checks can happen on the conceptual descriptions of the asset model level and are implemented via constraint expressions (see section 3). As an example, consider:

```
class SoftwareModel {
   concept relationship   classes : ClassDescription*
           relationship   objects : ObjectDescription*
           relationship   sequences : Sequence*
           constraint     sequences.objActs.msgs.name
                          <= classes.operations.name
                          and ...; matching signatures
           ...
}
class Sequence {
   concept relationship   objActs : ObjectActivation*

}
class ObjectActivation {
   concept relationship   obj : ObjectDescription
           relationship   msgs : Message*
}
⋮
```

The constraint on `SoftwareModel` checks whether all messages used in sequence diagrams are available as operations in class diagrams. Of course, such constraints can also model inter-phase relationships, e.g., that every class in the conceptual model also needs to exist in the implementational one.

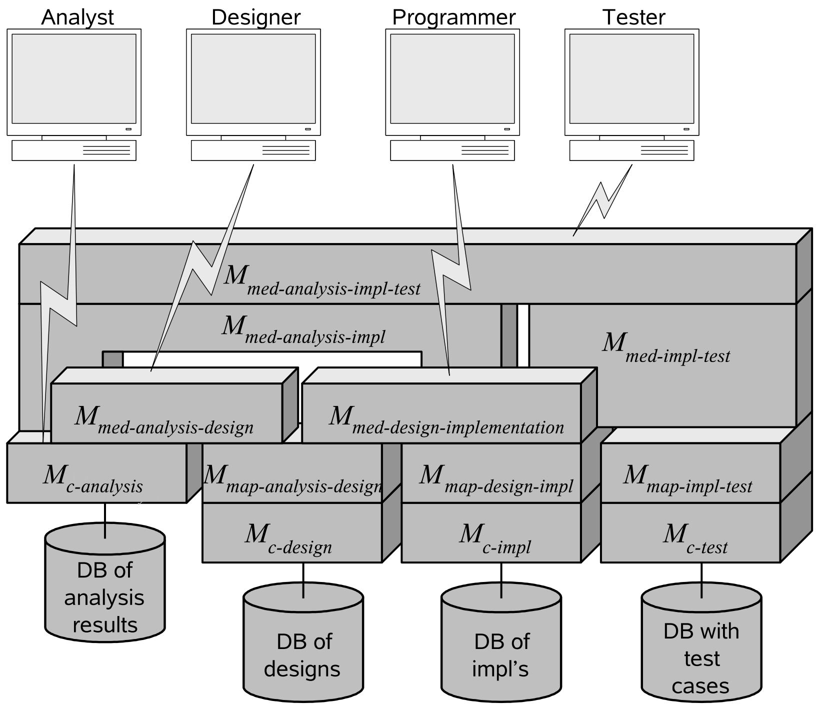

Analyst
Designer
Programmer
Tester
$M_{med\text{-}analysis\text{-}impl\text{-}test}$
$M_{med\text{-}analysis\text{-}impl}$
$M_{med\text{-}impl\text{-}test}$
$M_{med\text{-}analysis\text{-}design}$
$M_{med\text{-}design\text{-}implementation}$
$M_{c\text{-}analysis}$
$M_{map\text{-}analysis\text{-}design}$
$M_{map\text{-}design\text{-}impl}$
$M_{map\text{-}impl\text{-}test}$
$M_{c\text{-}design}$
$M_{c\text{-}impl}$
$M_{c\text{-}test}$
DB of analysis results
DB of designs
DB of impl's
DB with test cases

Fig. 4. A CCMS configuration for a software engineering scenario

Violation of constraints can be reported for the system model as a whole, resulting in an integrated issue-list for the complete system. Some types of changes done by developers can result in violations of constraints (e.g., the renaming of a class in the transition from conceptual to implementational class diagram). This will show up in the issue list and can be clarified by the developer. The clarification serves a double purpose: It resolves what seemed to be a violation, but it also creates a link between two entities whose connection the system could not have detected automatically.

Fig. 4 shows a configuration for the management of SE entities with interrelationships. It is structurally similar to that of fig. 3. In the example there is a database for every phase of a typical SE process. According to the architecture of CCMSs there is a client module for each database. In conventional tool settings users in each phase work with exactly one of the databases at a time. In the example of fig. 4 this is the case for the analysts who store their results in the DB of analysis results.

For later stages the shown configuration supports the linking of contents as explained above. E.g., designers store their artefacts into the design database. To additionally relate their results to those of the analysis phase they do not work directly with the module $M_{c-design}$ which exclusively accesses the design database. Instead, they work with a mediation module through which they access both the analysis and the design databases.

The intermediate mapping module $m_{map-analysis-design}$ extends design assets such that relations to analysis assets are added. This way, designers can establish links between artefacts from analysis and design phases. These links can later (see section 4.2) be exploited to support a variety of functions.

Mediation modules do not only bridge the gap between analysis and design. They can also mediate between the other phases (see fig. 4). Programmers work on capturing the results of the design phase in actual code. In traditional environments, they would look at the design documents and then work exclusively with the implementation database. This approach breaks the links between design and implementation artefacts. Again, by use of a mediation module, this problem is resolved.

Testers need access to an even wider selection of data. They do not only work on the test database, but on a mediation module that also accesses the analysis and implementation phase components. This mediation module allows for seamless navigation through all the artefacts along the preserved links.

Obviously, the asset model supports linking of content between the various phases of the development process. Such links can be used to establish traceability [8, 15]. Thanks to the overarching conceptual modelling, content from different phases is clearly connected along the lines of the corresponding concepts. These links can explicitly be modelled (as in approaches like "GRIDS" [30]) in the asset model (see section 3), but mostly this is not necessary, as the conceptual information persists across phase boundaries. With this model-inherent support of CCMSs it is possible to achieve concise semantic connections of content. This was previously very difficult and thus usually not attempted [9].

4.2 Applications and Application Support

A CCMS like the one outlined in the previous section can be used to support all aspects of the software development workflow. In this section, we will briefly introduce some interesting use cases.

Navigation through conceptual linking of artefacts. Through mediation modules that integrate various client modules, users are enabled to navigate along any path, even if it spans across multiple phases. This is beneficial for anybody taking part in software development to find artefacts that are related. A prominent example are implementers, who can now easily access the documentation for the code at hand. The same mechanisms allows them to reach back into design or even analysis to retrieve documents which concern the software entity at hand. This way it is much easier to understand what the entity does and why it is there. Especially the backwards links ("why do we need this class?") are not obvious in traditional software development. A CCMS is able to give a detailed account of the requirements and design decisions that lead to the existence of the entity in the implementation phase.

Custom perspectives for different roles. By means of personalised models different user roles can be provided with customised views on the development artefacts, e.g., to work on UML class diagrams with or without attributes shown. This way, users have to deal only with data which is relevant to their task.

Cross-phase constraints. Maintaining consistency between the artefacts of several development phases is a major problem in software development. Through the conceptual links that are also used for navigation, one is able to write constraints which span phase boundaries. An example of this was given in the previous section, but further use cases are not difficult to imagine: Tracking of changes from design to implementation (and the other way around), ensuring test coverage of analysis requirements, or monitoring the degree of completion of the implementation with respect to design documents.

Transparent distribution. Modern development happens in teams. This calls for remote cooperation of all the members of a team. In particular, they need to share a common information basis to ensure that the created artefacts are consistent with each other. Moreover, all the functionality outlined above needs to be available across several systems in a concurrent and transparent way. CCMSs support this by means of distribution modules (see fig. 2). In combination with the personalisation abilities, distribution in CCMSs goes beyond of what is traditionally used in software development. Through personalisation, users cannot only work on a common data base in a distributed fashion, but are also permitted to deviate from the community for some time and then remerge their artefacts. This can e.g. be used for branching source code.

5 Summary and Task Outlook

In this paper we present our asset-oriented information model as a conceptual content model gained by generalising the notion of functional abstraction predominant in the computational domain. Its safe and efficient use for the management of computational content is one of the fundamentals of state-of-the-art software development tools.

The presented generalisation step towards domain-independent conceptual content modelling makes CCMSs sound candidates for supporting the entire SE process ranging from application domain entities via artefacts for software design and architecture down to computational entities for system implementation and execution, thereby improving the coherence of SE processes.

Future work will address the extension of asset-based models for SE processes. This will include the development of models for the various development steps and their associated activities. Essentially, we address two goals: First, process portals can be built that collect examples of "best-practice" processes or parts thereof [25] which sometimes are more easily judged than abstract descriptions [17]. Second, making use of openness and dynamics allows the individualisation of software development processes to better suit a project's pragmatics [6].

Extending conceptual support beyond the traditional phases of software development into runtime will make information about the development of the system available during system execution. All sorts of services including debuggers and other inspection services will benefit from such extended information that allows to trace entities back to, e.g., the analysis phase.

Obviously, integration of CCMSs with traditional software development environments is essential in practical use. We expect that due to the modular architecture of CCMSs we will be able to create bridges to specific tools and representations (such as XMI based ones) which partially automate the conceptual modelling task of the developers. Also, with MDA [12] receiving much research interest lately, we will investigate how (semi-) automatic transitions between various development phases can benefit from the use of conceptual modelling.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors. *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. Topics in Information Systems. Springer-Verlag, 1984.
3. Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-based Design. In *Proceedings of the First International Workshop on Embedded Software*, volume 2211 of *LNCS*, pages 148–165. Springer-Verlag, 2001.
4. Alexander Egyed and Nenad Medvidovic. A Formal Approach to Heterogeneous Software Modeling. In *Proceedings of the Third International Conference on Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*, pages 178–192, 2000.
5. Hartmut Ehrig, Bernd Mahr, Felix Cornelius, Martin Groe-Rohde, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag, 2nd edition, 2001.
6. Martin Fowler. *UML Distilled*. Addison-Wesley, 3rd edition, 2003.
7. Andreas Gawecki and Axel Wienberg. Report on the Tycoon-2 Programming Language. Version 1.0 (Draft). Technical report, Higher-Order GmbH, Hamburg, and Software Technology and Systems Institute, Hamburg University of Science and Technology, 1998.
8. Orlena C.Z. Gotel and Anthony C.W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *First International Conference on Requirements Engineering (ICRE)*, pages 94–101. IEEE Computer Society Press, April 1994.
9. Duane Hybertson. Strengthening the Modeling Foundation of the MDA. In *Workshop in Software Model Engineering*, 2002.
10. Florian Matthes. Higher-Order Persistent Polymorphic Programming in Tycoon. In *Fully Integrated Data Environments*, ESPRIT Basic Research Series, pages 13–59. Springer-Verlag, 2000.
11. Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology*. Kluwer, 2000.
12. Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical Report omg/2003-06-01, OMG, June 2003.

13. Mark A. Musen. Ontology-Oriented Design and Programming. In *Knowledge Engineering and Agent Technology*. IOS Press, 2000.
14. Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice-Hall, 3rd edition, 1996.
15. Balasubramaniam Ramesh and Matthias Jarke. Toward Reference Models of Requirements Traceability. *Software Engineering*, 27(1):58–93, 2001.
16. György Revesz. *Lambda-Calculus: Combinators, and Functional Programming*. Number 4 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.
17. Thomas Rose, Martin Fünffinger, Holger Knublauch, and Christian Rupprecht. Prozessorientiertes Wissensmanagement. *Künstliche Intelligenz*, 16(1):19–24, 2002.
18. Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), 1977.
19. Joachim W. Schmidt and Florian Matthes. The Rationale behind DBPL. In *3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems*, volume 495 of *LNCS*. Springer-Verlag, 1991.
20. Joachim W. Schmidt and Hans-Werner Sehring. Conceptual Content Modeling and Management: The Rationale of an Asset Language. In *Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 469–493. Springer, 2003.
21. J.W. Schmidt, H.-W. Sehring, M. Skusa, and A. Wienberg. Subject-Oriented Work: Lessons Learned from an Interdisciplinary Content Management Project. In *Advances in Databases and Information Systems*, volume 2151 of *LNCS*, pages 3–26. Springer-Verlag, 2001.
22. Hans-Werner Sehring. *Konzeptorientiertes Content Management: Modell, Systemarchitektur und Prototypen*. PhD thesis, Hamburg University of Science and Technology (TUHH), 2004.
23. Hans-Werner Sehring and Joachim W. Schmidt. Beyond Databases: An Asset Language for Conceptual Content Management. In *Proceedings of the 8th East European Conference on Advances in Databases and Information Systems*, volume 3255 of *LNCS*, pages 99–112. Springer-Verlag, 2004.
24. German Shegalov, Michael Gillmann, and Gerhard Weikum. XML-enabled workflow management for e-services across heterogeneous platforms. *VLDB Journal*, 10(1):91–103, 2001.
25. Carla Simone and Monica Divitini. Ariadne: Supporting Coordination through a Flexible Use of the Knowledge on Work Processes. *Journal of Universal Computer Science*, 3(8):865–898, 1997.
26. Ian Sommerville. *Software Engineering*. Addison-Wesley, 2000.
27. Ragnhild van der Straeten. Semantic Links and Co-Evolution in Object-Oriented Software Development. In *Proc. 17th IEEE International Conference on Automated Software Engineering*, page 317. IEEE Computer Society, 2002.
28. S. White and C. Lemus. Architecture Reuse Through a Domain Specific Language Generator. In *Proceedings of the Eighth Workshop on Institutionalizing Software Reuse*, 1997.
29. G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25:38–49, 1992.
30. Andreas Zamperoni. GRIDS – graph-based, integrated development of software: integrating different perspectives of software engineering. In *Proceedings of the 18th International Conference on Software Engineering*, pages 48–59. IEEE Computer, 1996.