Sebastian Bossung, Hans-Werner Sehring, Patrick Hupe and Joachim W. Schmidt. “Open and Dynamic Schema Evolution in Content-intensive Web Applications”. In Proceedings of the Second International Conference on Web Information Systems and Technologies, pp. 109-116, April 2006

OPEN AND DYNAMIC SCHEMA EVOLUTION IN CONTENT-INTENSIVE WEB APPLICATIONS

Sebastian Bossung, Hans-Werner Sehring, Patrick Hupe, and Joachim W. Schmidt
Hamburg University of Technology
Hamburg, Germany
http://www.sts.tu-harburg.de
{*sebastian.bossung,hw.sehring,pa.hupe,j.w.schmidt*}*@tuhh.de*

Keywords: web information systems, module-based architecture, schema evolution, software generation, case study.

Abstract: Modern information systems development is a complex task for it must fulfill a large variety of application- and architecture-oriented requirements. Furthermore, such requirements often are a moving target for the developer, not only because the system has to stay *open* to a constantly changing application domain, but also because new requirements are added during the extremely long lifetime of such information systems. To make things worse, modern information systems are operated in a 24x7-modus which generates the pressure of highly *dynamic*, almost online system evolution.
A main source of problems such development projects struggle with originates from the lack of a systematic subdivision of large software systems into manageable modules. As a consequence developers are traditionally involved in a complex patchwork of manual efforts to keep the various parts of the system in sync with each other and with the system’s requirements.
In this paper we outline our approach to information system development which is based on a model for *Conceptual Content Management* (CCM). Our CCM approach profits from the dynamic, model-driven generation of smaller modules, which can be combined automatically into the full system. The generation process uses a CCM model of the application domain(s) from which our *compiler framework* dynamically generates the schema-dependent parts of the system. Due to the dynamic nature of this generation process, we are able to provide adequate support for both schema evolution and personalization of such a system. We have successfully employed the CCM approach to the development of complex web information systems. We give a brief account of CCM development and present an application example.

1 INTRODUCTION

Since a large number of heterogeneous requirements have to be met by today’s information systems, such systems tend to become very complex and hard to manage. As a result software adaptations become increasingly difficult to handle as the systems age. However, change—specifically change in the application’s schema space—is a fact that developers have to live with. Unfortunately, each evolution of the schema usually impacts almost all parts of the system, resulting in time-consuming and error-prone adjustments.

This paper outlines a *generative* approach to system development that helps to manage model evolution as well as system adaptation through three kinds of contributions:

- Content schemata are raised to the more abstract level of a conceptual model for the application.
- Systems are composed of carefully designed smaller modules, which can be combined freely by means of a uniform API.
- System modules are dynamically generated from conceptual domain models enabling the system to react quickly to schema evolution by recombining modules without manual intervention.

In fact, the system can react to such changes automatically and almost in real-time, allowing users to change the schema according to their needs (see section 3 for more details on schema evolution). Such schema changes—or schema personalisations—would not be possible in a manually implemented system. We call such systems Conceptual Content Management Systems (CCMSs) and the approach Conceptual Content Management (CCM).

In the CCM approach entities are modelled in an object-oriented way by so-called Assets which define the (multimedia) content of the entity together with

its characterising attributes (primitive values and relationships) as well as its constraints. An Asset Definition Language (please refer to (Schmidt and Sehring, 2003) for details) is applied to define the conceptual schema of the application domain. The conceptual Asset schema is then used to generate modules from which the application is composed. These modules have a uniform interface and provide—amongst other things—for a separation of concerns. As a consequence, modules can be arranged freely enough to enable the systems to support open and dynamic asset schema evolution. Note that it is important that content remains accessible across schema changes, even if these are just personal ones for an individual user.

Extensive practical experience with the CCM approach has proven quite successful, as we were able to deliver running systems rather quickly. These systems can keep up with schema evolution, a necessary demand despite an extensive and careful phase of application analysis.

The remainder of this paper is organised as follows: We first give a motivation for our work along with a more detailed description of the problem space. This is followed by a brief overview of schema evolution, before we outline in section 4 how CCMSs are implemented. We critically discuss the CCM approach and future research directions in section 5 and conclude with a summary in section 6.

2 MOTIVATION

Web applications are typical representatives of systems which are built according to a layered architecture. The domain model on which they are grounded determines the peculiarities on each layer: on the data layer one needs to be able to store all information required to enable all tasks of the application and presentation layers. Information includes content and characterising attributes of domain entities as well as management information.

On the presentation layer such information is represented based on the data, and user input is accepted to navigate through the data, as well as to create, manipulate, or delete data.

The application layer mediates between data storage and presentation tasks. In web-based systems it typically performs only few computations, if any.

All layers of web applications are highly dependent on a conceptual model describing the application domain. Furthermore, the layers are highly interrelated. Despite the layering, changes applied to one layer often impact the whole system, making its lifecycle difficult to handle. Though some argue that web-based systems do not really evolve, we aim to support dynamic schema evolution and personalisation, which also raise the lifecycle issue. Our CCM approach to evolving, personalisable systems (see section 4) is based on the generation of all the layers from a rich conceptual model.

In the following subsections we discuss the generation of the layers of a web-based information system.

In section 5.2 we report on a particular project which raised additional requirements. The usage scenario demanded for offline data input through rich clients with later synchronisation with the main data pool. A diverse publication process needed to guarantee data quality and the enforcement of domain-specific data protection. The latter has two aspects: on the one hand, the nature of the data requires means to protect personal rights which might be infringed by the uncontrolled publication of data. On the other hand, contributing researchers demand for a controlled publication of their intellectual property, especially carefully choosing the point in time at which their data should become available to others.

2.1 Data Storage

To fulfil the requirement of providing fast access to large amounts of content, well established databases are used on the data layer. Still, the data model has to be distinguished from the conceptual model which lead to its creation. Usually, many commitments have to be made to express a model in a data definition language, depending on the paradigm a database is following. Nevertheless, input and output language of a web-based system are the domain-specific language meaningful to the users.

If the application domain is conceptually modelled in a language that is more expressive than a data definition language, the database schema can be fully generated to match this conceptual model.

Support for personalisation calls for the ability to perform schema evolution. Schema evolution is well understood at the data level, see section 3. Therefore, the generation of the data layer of a web application does not lead to severe problems.

Another aspect associated with the data layer is the formulation of application-level constraints. Though these go hand in hand with the application layer and thus are often implemented there, good reasons exist to handle them in the lowest possible layer (Date, 2000). Therefore, constraints formulated by users in a conceptual model need to be translated to constraints on the data model.

2.2 Web Presentation

Presentation is an indistinguishable part of web applications. The user interface needs to reflect the current set of data with respect to the domain model.

Web pages generated to represent information and to interact with the user exhibit all the problems of user interface design. Therefore, they require the consideration of more than just the conceptual model, namely issues of usability. Additional requirements of generated web pages are localisation (language, output format of dates, character encodings, etc.), context dependencies (the role of the user viewing a web page, etc.) and many more.

Still, our experiments have shown that due to the well-understood, highly uniform presentation paradigms of the web, and users who are accustomed to these, most of a web application can be generated automatically: tabular representations of data, form-based input and manipulation of data, etc.

Generated templates for web pages appear to be acceptable, though not everything is most efficient (in terms of the user experience). Therefore, the generation of web pages is directed by generation hints.

2.3 Application Logic

The application layer of web-based information systems usually comes down to a mapping between the users' conceptual perspective and the data model, accompanied by some checks performed on data input and simple data transformations. Since checks can be expressed as constraints on the data layer, the application layer of a web-based information system basically plays the role of a controller according to the model-view-controller pattern (Collins, 1995).

To this end, application logic can be generated from a conceptual model and its constraints since it merely makes generic use of the data layer that has been generated accordingly.

The mapping task of the application layer becomes a bit more complex if there is more than one subsystem on the data layer. More complex systems will use more than one database, e.g. one for structured data and another for multimedia content, or one for application data and another for management data. Special web applications like portals or brokers even use complete third-party systems as their data sources.

In such cases, more than the knowledge of the respective models is needed. When providing uniform access to different subsystems, the protocols for their use have to be considered. Since theses protocols usually differ to a large degree, some application-specific wrapping code might have to be supplied. Such specialties are recognised by the architecture of the systems we generate (see section 4.3).

2.4 Complex User Interfaces

More complex user interfaces than those supplied by web pages can be found outside the layers forming the architecture of a web application. Especially, client software with a rich user interface often supports tasks performed by users in special roles which differ from that of the occasional web user.

Examples are content editors of content management systems, catalogue software, etc. In our case, an offline client, which allows content entry even when not connected to the internet, was needed.

Such clients are typically fat clients which contain application logic of their own. Their application logic provides support for specialised roles and exceeds the behaviour of the application layer of a web-based information system.

Therefore, there is no uniform way of generating application logic from the conceptual model alone. It needs to be amended by a specification of the task the client software must support. Still, parts can be generated the same way as the layers of a web application. The arbitrarily complex functionality of the whole client software has to be defined by means of programming which can be based of these parts.

A fat client might also contain a data store on its own, as is the case for the client for offline data input. Such a data store is again generated from the conceptual model, possibly adding a different set of management data. At this point we had the chance to reuse modules. Furthermore, employing a common conceptual model made it possible to distinguish between application data and management data, thus clarifying the synchronisation task between offline clients and the web application.

3 SCHEMA EVOLUTION

As pointed out above, many applications live in a world of constant change. This applies particularly to the underlying schema, as it needs to be adapted to ever changing requirements. Traditional information systems employ the same schema for all users, such that all data need to be lifted to the new schema. Problems and possibilities of such actions are well understood on the database level, as witnessed by extensive literature on the subject, see e.g., (Roddick, 1992), (Lerner and Habermann, 1990). Severe problems arise when data need not only be lifted to the new schema, but a backwards path is also required, compare for example literature on updatable views (Kuno and Rundensteiner, 1995; Abiteboul et al., 1998).

In the case of CCMSs, evolution of the general application schema is one concern. Such evolution is necessary—as in other types of systems—if the model of the domain needs to be changed. Examples include modelling errors, where the domain was only fully understood during the course of the project, and extension of the scope of the system, when new data

Figure 1: Six kinds of modules

need to be recorded. In addition CCMSs acknowledge that the user community might not agree on the appropriate schema for the domain. Therefore, each user (or group of users) is permitted to modify the schema according to their personal opinion. We call this (schema) personalisation. As many such personalisations might occur in parallel, the system generally has to cope with many coexistent schemata.

Note that an important difference between schema evolution and personalisation lies in the fact that personalisation will always require a backward path for the data. While users might wish to personalise their schema, they still want to collaborate with fellow users which involves the need to exchange data. However, these fellow users might work with different schemata. In many cases of schema evolution one can make do without such a backward path (e.g., disallowing write access for users of the "legacy" schema is often feasible).

Additionally, collaboration in the presence of schema personalisation might also require schema mapping (see, e.g., (Rahm and Bernstein, 2001)), where data have to be converted between schemata which were developed independently of each other.

4 REALIZATION

In this section we give details on our generative approach to system implementation.

4.1 Modules

CCMSs are composed of *modules*, which have a uniform interface and can thus be freely combined. We have identified six kinds of modules, which are depicted in figure 1. The six kinds of modules are: (1) server modules, which provide external services, (2) client modules, which lift underlying systems (e.g., databases) to the module API, (3) mediation modules, which combine two modules—where the two possibly have different roles, (4) hub modules, which do so for many modules, but all modules have the same role, (5) transformation modules, which transform from a base to a target schema, and (6) distribution modules, which provide for cross-system communication. Each module builds on several other modules (base modules) and provides the uniform module API (indicated by dark bars in figure 1) to its super modules. See (Schmidt and Sehring, 2003) for further details.

The module interface provides standard operations, such as creation, modification and deletion of instances, as well as their retrieval. Further details are available in (Sehring and Schmidt, 2004).

Modules are combined into *components* using the mediator architecture as proposed by (Wiederhold, 1992). Modules are stacked in layers, such that the module *configuration* is a directed, acyclic graph. Components provide a number of services to their modules, including identifier resolution. The requirements of the application are implemented by combination of modules, see section 4.4 for some combination patterns.

4.2 Compiler Framework

In our approach, the diversity of modules is reflected by generators. There are individual generators for all variations of a module kind. Knowledge about database management systems is for example implemented in a generator that creates client modules which are based on a relational database.

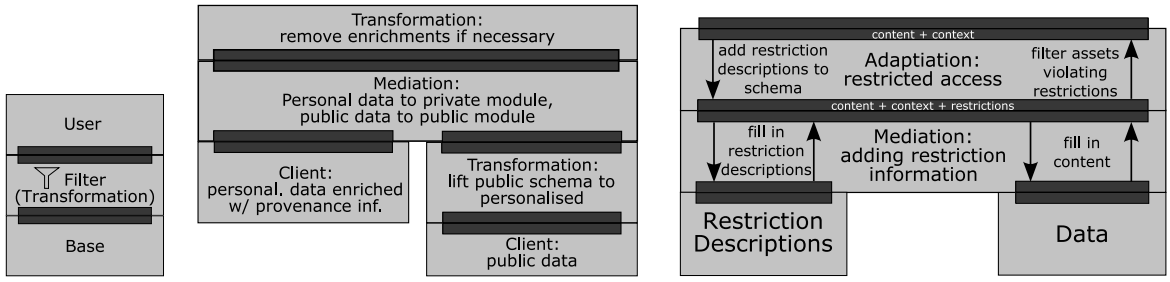
Writing generators is a complex task, mainly because setting up an infrastructure for them (Smaragdakis and Batory, 1996) is difficult. Therefore, our model compiler for CCM is designed as a framework with generators as extension points. In conjunction with a facility for code generation it constitutes a domain-independent meta-programming infrastructure (Smaragdakis et al., 2004).

A complete CCMS is generated by an instance of the compiler framework, using an appropriate set of generators. The framework controls the overall compilation process following the typical structure of compilers consisting of frontend and backend components, where generators form the backend.

The execution of generators is scheduled based on their interdependencies, which are computed by the framework based on an extended notion of symbol tables. Theses symbol tables are also used for communication between the generators. A client module for access to a, for example, relational database needs both the database schema, which is produced by a separate generator, and the uniform module API, which is created from the conceptual domain model by a central API generator. Both schema and API definitions are provided by means of symbol tables.

4.3 Implementing Specialties

Most applications will have a few requirements which cannot be realised by generators. This may for ex-



(a) Filter (b) Schema evolution (c) Restricted access

Figure 2: Module patterns

ample be the case when a new type of requirement comes up for the first time. One could then implement this requirement on top of the generated system. If the hand-written code has anything to do with the schema, this renders personalisation impossible. A better approach is to work on generators to include the requirement and be able to reuse the generator in future implementations. We have identified three options to realise new requirements in a CCMS:

1. **Writing a new generator.** This usually makes sense if one is dealing with totally new requirements, e.g., a new technology is to be incorporated into the CCMS. In unusual situations one might choose to implement a custom generator just to enable personalisation for this particular application. This can be worth the effort if heavy use of personalisation is expected.
2. **Subclassing an existing generator.** This is possible in special cases, most prominently if the new requirement is an extension of requirements handled by an existing generator. It does not seem to be a good idea to implement orthogonal requirements this way as is this would require hook points as proposed by (Aßmann, 2003) for components.
3. **Achieving the desired effect through module combination.** Of course, not all of the required modules will be available, but it might be possible to write small generators that create the missing modules. Using modules of the new kind, one can then combine existing and new modules into a system that meets all requirements. For an example see the blocking module in section 5.2.

4.4 Module Patterns

During system development we have identified several patterns of module combination. These help to structure complex systems that can be built from a high number of individual modules (see, e.g., figure 3). We present four of them in this section, the first three are depicted in figure 2.

Filter: This pattern is a basic building block found in most systems. Instances from a Base module are filtered according to certain criteria and only those passing the filter are visible to the User module. Filter criteria can be manifold (e.g., offering only recent instances in news applications).

Schema Evolution: Many of our systems require schema evolution or personalisation (section 3). In figure 2(b) modules are combined such that users of the top-most transformation module have uniform access to instances according to their personal schema. This is achieved by first transforming from public to private schema, then combining instances (the mediation module also keeps track of personalised instances), and finally projecting away provenance information that is necessary for personalisation.

Restricted Access: In this pattern, two pools of data are kept in separate client modules: application Data and Restriction Descriptions. They are combined by a mediation module. Users of the Adaptation module can pose queries also containing information that is necessary for restriction enforcement (e.g., user or group information). Based on this information, the appropriate descriptions for restricted access as well as application data are retrieved. The adaptation module finally filters the retrieved application data according to the provided restriction descriptions. Optimisations include not posing a query on application data, if it can be determined from the restriction descriptions that none of the retrieved instances will be accessible (e.g., if a user does not have read permissions for instances of a certain class).

Fat/Thin Client: By placing modules on different components, fat or thin clients can be modeled (not shown in figure 2, see (Bossung et al., 2005)).

5 DISCUSSION

In this section we discuss some general benefits of the CCM approach in the context of our application

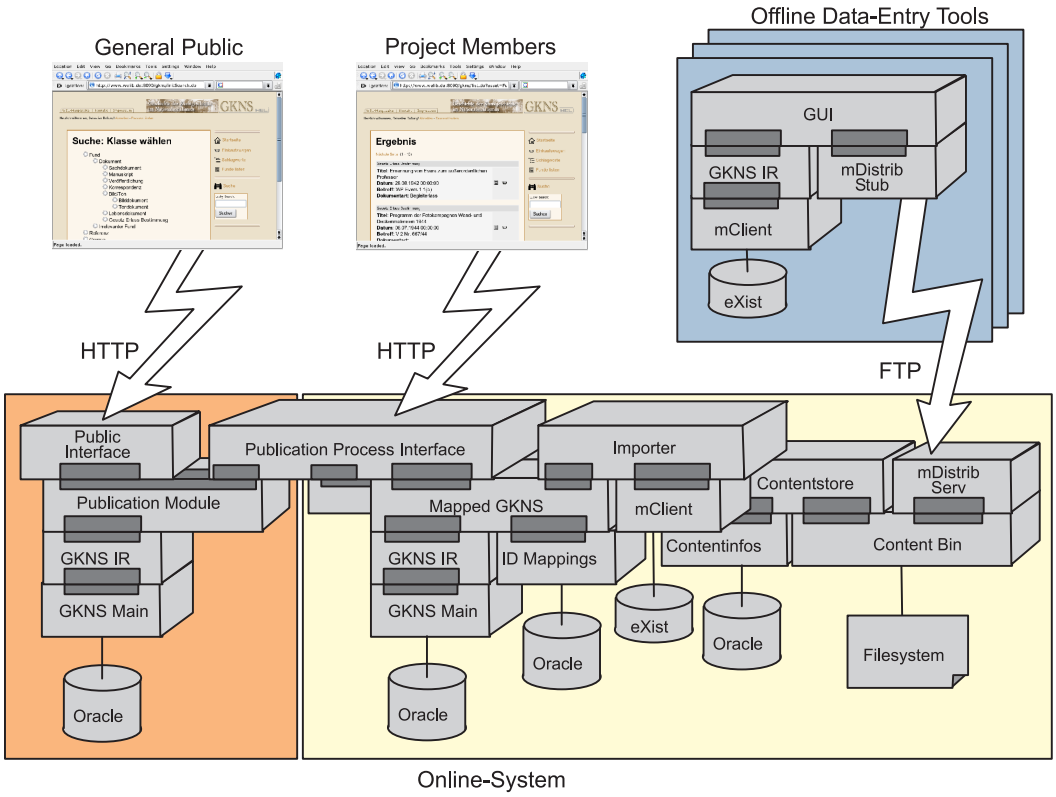


Figure 3: A system composed of modules. Some of the modules (e.g., the “Importer” or the “Publication Process Interface”) are actually marcos which represent whole patterns of modules (see section 4.4).

projects. We also consider open issues and future research directions.

5.1 General Benefits

The CCM approach of combining conceptual modelling with system generation results in the following specific benefits:

Schema evolution: The CCMS architecture supports schema evolution on a per-user or per-group basis. Traditional schema evolution is a special case where the schema for the entire application evolves.

Separation of concerns: Each module addresses a specific system functionality and can only use its direct base modules. This way, the distribution of functionality in the application as well as dependencies between modules are made explicit. As a result it is easy to subdivide the system along these interfaces to insert new functionality (e.g., a module which encapsulates full-text search).

Reuse: There are two cases of reuse in CCMSs: (1) Generated modules can be used in multiple places, and (2) implementation knowledge is reused by generators. The first case is made feasible by the uniform module API which enables module combination in a variety of ways. The second helps to avoid errors which are frequently made when implementing modules manually. Generating modules instead of coding them also means that they are implemented with the expertise of the generator author which can usually be assumed to be above average.

Development speed-up: Once the appropriate generators are written, implementing new systems or reimplementing existing ones (to meet new requirements) is much quicker. In our experience development of generators soon paid off. We do not have exact figures but our estimate is that the implementation of the generator is about 2 to 10 times as costly as a manual implementation of the corresponding module and, therefore, provides a positive return on investment after about half a dozen module generations.

5.2 Project Experience

The application described here is a complex, multi-user application[1], as depicted in figure 3. Its key purposes are the storage of (scans of paper-) documents, the collection of information about these documents

[1] See http://www.welib.de/gknsapp

and their interrelationships. Documents are obtained from archives, where direct internet access is usually not provided. The application system, therefore, was divided into two parts: a central system, which integrates all content that has been entered through the offline tool and uploaded into the central system. As the uploads can be quite high in volume, the central system imports them asynchronously.

The central system is subdivided into two parts: An editorial system where uploads are double-checked and a public system where quality-assured data is made available to registered users.

The entire system is composed of modules, most of which are generated; some are schema independent and only a few are implemented manually. Data is stored in client modules which are backed by either a light-weight XML database in the offline tool or a standard relational database.

Through the use of modules we were able to reuse large parts of the system in several places. The client module for the XML database is used in the content entry tool as well as in the central system for importing data from the filesystem; both modules are identical. Another module used in both parts of the system is the full text search module GKNS IR. The central system uses the client module ID Mappings to keep track of imported data.

As pointed out in section 2.1, full generation of client modules is usually possible and indeed all client modules in this application are fully generated. As pointed out in section 2, generation of user interfaces is more difficult. The content entry tool contains a few optimised workflows which were largely written manually, albeit still personalisable. The personalisable web interface of the editorial system is simpler and could therefore be generated to a large degree.

The client module which provides XML database persistence was first implemented manually, because an appropriate generator did not exist and quick delivery of the content entry tool was crucial to the project schedule. Later, a generator for this module was written to support schema evolution. The implementation of the generator took about twice as long as the manual implementation of the client module for a schema of small complexity (ca. 20 classes).

The entire system uses all of the patterns discussed in section 4.4. Not all are, however, shown in figure 3. There is for example one special requirement in the project that made the use of a filter module in both the public and the editorial web interfaces necessary. Since some application content is quite sensitive, public access has to be restricted on a per-instance basis for a configurable time span. We have, therefore, inserted a filter module, which implements this restriction by essentially enriching each query with an additional constraint, which in turn checks whether restrictions apply.

5.3 Open Issues

At the current state of development CCM still has a number of open issues. These do not limit the power of the approach in general, but cause practical problems in mainly two areas:

User interfaces: User interfaces contain numerous special cases (such as custom layout, naming or error handling) which require some manual extension of the generated skeleton.

Managing personalisation: While users should be able to personalise the schema according to their needs, they should also be able to work together collaboratively, as this is one of the key purposes of information systems (Goldin and Thalheim, 2000). However, when applying personalisation, users can make schema changes which effectively cut them off from other users. To be "cut off" means having to modify each personalised data instance manually when it needs to be shared with others. Cases of this include eliding mandatory attributes or modifying attributes with functions that do not have inverses.

5.4 Outlook

We will focus future work in three areas: Generating user interfaces, formalising personalisation, and identifying further module patterns relevant to information systems.

As pointed out, user interface developement often deals with numerous special cases. We expect that user interfaces due to their irregularity will always require some manual work. We have identified four possibilities to influence the generation of user interfaces:

- Changing the conceptual model of the domain;
- Configuring the generator (e.g, with a domain specific language for user interfaces);
- Passing parameters to the component of the generated system;
- Hand-coding parts of the user interface on top of the generated code. However, the hand-coded parts must be invariant to the schema, otherwise one loses personalisation.

Our current generator for web applications combines all four cases but its configuration language is still weak. There exist modelling paradigms for rapid prototyping of user interfaces (e.g., SiteLang proposed in (Thalheim and Düsterhöft, 2001)) and for constructing navigational models such as (Valderas et al., 2005). Combining such approaches with system generation will enable a high degree of customisation while still fully supporting schema personalisation.

Furthermore, schema personalisation should not prevent users from cooperating (i.e., exchanging their

content) with others. Thus, in personalisable systems changes have to be tracked to warn users of situations in which their schema personalisation steps disconnect them from their community. State-based typing of domain entities (e.g., (Strom and Yemini, 1986) and (Bierhoff and Aldrich, 2005)) can help to build models for personalisation.

In addition, it will be interesting to identify further module patterns. As more applications are developed using CCM, the understanding and support of common application issues will improve. We plan to build a library of such patterns.

6 SUMMARY

In this paper we have presented our Conceptual Content Management approach to compose complex information systems from smaller, generated modules. The generation is based on a CCM model used as input to the compiler framework. CCM has a number of benefits: Due to the highly structured nature of such systems and the possibility of dynamically generating new modules, systems are able to substantially support schema personalisation and evolution. CCM facilitates reuse of code as modules are freely combinable. Knowledge about system implementation is codified in the generators and can be reused in different project instances and phases. Larger real-world projects have shown that the proposed approach works well in practice. Even though problems remain—mainly in generating user interfaces and in integrating of manually written system code—system generation has enabled us to deliver quickly complex information systems in the context of changing and evolving domain models.

Acknowledgements. We would like to thank our colleagues of the art history departments in Berlin, Bonn, Hamburg, and Munich for providing us with many insights in the application projects. Furthermore, we would like to thank Deutsche Forschungs Gemeinschaft (DFG) for its grant on the WEL-GKNS project.

REFERENCES

Abiteboul, S., McHugh, J., Rys, M., Vassalos, V., and Wiener, J. L. (1998). Incremental maintenance for materialized views over semistructured data. In Gupta, A., Shmueli, O., and Widom, J., editors, *VLDB'98*, pages 38–49. Morgan Kaufmann.

Aßmann, U. (2003). *Invasive Software Composition*. Springer-Verlag.

Bierhoff, K. and Aldrich, J. (2005). Lightweight object specification with typestates. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 217–226.

Bossung, S., Sehring, H.-W., and Schmidt, J. W. (2005). Conceptual content management for enterprise web services. In *Perspectives in Conceptual Modeling: ER 2005 Workshops*, volume 3770 of *LCNS*, pages 343–353. Springer-Verlag.

Collins, D. (1995). *Designing Object-Oriented User Interfaces*. Benjamin/Cummings Publishing.

Date, C. (2000). *What Not How – The Business Rules Approach to Application Development*. Addison-Wesley.

Goldin, D. Q. and Thalheim, S. S. B. (2000). IS = DBS + Interaction: Towards Principles of Information System Design. In *Proc. ER 2000, 19th Int. Conf.*, volume 1920 of *LNCS*, pages 140–153. Springer-Verlag.

Kuno, H. A. and Rundensteiner, E. A. (1995). Materialized object-oriented views in multiview. In *RIDE-DOM*, pages 78–85.

Lerner, B. S. and Habermann, A. N. (1990). Beyond schema evolution to database reorganization. In *OOPSLA/ECOOP '90*, pages 67–76, New York, NY, USA. ACM Press.

Rahm, E. and Bernstein, P. A. (2001). A Survey of Approaches to Automatic Schema Mapping. *The VLDB Journal*, 10:334–350.

Roddick, J. F. (1992). Schema evolution in database systems: an annotated bibliography. *SIGMOD Rec.*, 21(4):35–40.

Schmidt, J. W. and Sehring, H.-W. (2003). Conceptual Content Modeling and Management. In *Proc. Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 469–493. Springer-Verlag.

Sehring, H.-W. and Schmidt, J. W. (2004). Beyond Databases: An Asset Language for Conceptual Content Management. In *Proc. 8th East European Conf. ADBIS 2004*.

Smaragdakis, Y. and Batory, D. (1996). Scoping Constructs for Program Generators. Technical Report CS-TR-96-37, Austin, Texas, USA.

Smaragdakis, Y., Huang, S. S., and Zook, D. (2004). Program generators and the tools to make them. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 92–100. ACM Press.

Strom, R. E. and Yemini, S. (1986). Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171.

Thalheim, B. and Düsterhöft, A. (2001). Sitelang: Conceptual modeling of internet sites. In *Proc ER01*, volume 2224 of *LNCS*, pages 179–192. Springer Verlag.

Valderas, P., Fons, J., and Pelechano, V. (2005). Transforming Web Requirements into Navigational Models: An MDA Based Approach. In *Proc. ER05*, volume 3716 of *LNCS*, pages 320–336. Springer Verlag.

Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *IEEE Comp.*, 25:38–49.