Content is Capricious: A Case for Dynamic System Generation

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

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

Abstract. Database modeling is based on the assumption of a high regularity of its application areas, an assumption which applies to both the structure of data and the behavior of users. Content modeling, however, is less strict since it may treat one application entity substantially differently from another depending on the instance at hand, and content users may individually add descriptive or interpretive aspects depending on their knowledge and interests. Therefore, we argue that adequate content modeling has to be *open* to changes, and content management systems have to react to changes *dynamically*, thus making content management a case for dynamic system generation.
In our approach, openness and dynamics are provided through a *compiler framework* which is based on a conceptual model of the application domain. Using a *conceptual modeling language* users can openly express their views on the domain's entities. Our compiler framework dynamically generates the components of an according software system. Central to the compiler framework is the notion of generators, each generating a particular module for the intended application system. Based on the resulting *modular architecture* the generated systems allow personalized model definition and seamless model evolution.
In this paper we give details of the system modules and describe how the generators which create them are coordinated in the compiler framework.

1 Introduction

Most data-intensive applications serve, one way or another, as information systems (ISs) and call for some kind of persistence technology. High volumes of data and large user communities require additional functionality (query support, concurrency, recovery etc.) which nowadays comes nicely packaged as off-the-shelves database models and database technology.

Database modeling is rather strict in the sense that it is based on the assumption of a high regularity of its application areas. This assumption applies to both the structure of data and the behavior of users. Therefore, database models rest on a small set of agreed upon computational base types (numbers, strings, ...) and a few structuring mechanisms (mostly records and sets) used to design schemata shared by the entire application and its community. In an enterprise database, for example, the view on a company employee is defined before employee records are instantiated, and users of the database have to share the company's view.

Content modeling, however, is more capricious since it may treat each represented entity substantially differently depending on the instance at hand, and content users may individually add descriptive or interpretive aspects depending on their knowledge and interests. For example, when considering a particular piece of art, some users may be interested in the artist who created it, the material used and the prices achieved while others are more concerned about details on the period in which it was created, its meaning, etc.

Therefore, we argue that adequate content models have to be *open* to changes, and content management systems have to be *dynamic* to reflect such model changes. In other words, content management systems are seen as a case for dynamic system generation while database management system usually get away with the technically less ambitious case of generic system implementation.

ISs inherit the restrictions of fixed schemata and a uniform user community from the underlying database technology. The development of ISs usually accommodates to these restrictions: in an intensive phase of domain analysis the database schema is defined once and for all. Application logic and presentation are implemented with respect to the schema and the domain model. Because IS implementation relies heavily on certain schema information, later changes to this schema affect all parts of an IS, an aspect nearly prohibitive to any effort of dynamic system evolution or to any attempt of system personalization.

For content management systems such inflexibility cannot be tolerated. Content is viewed by users in different contexts with individual conceptual models in mind. Furthermore, users have to be able to define suitable models or adapt existing ones during the lifetime of a content management system. Therefore, model changes have to be integrated dynamically, without additional development steps which include manual intervention.

In our approach, openness and dynamics of content management systems is provided through a *compiler framework* which is based on a conceptual model of the application domain. In our *conceptual modeling language* users can openly express their views on the domain's entities, and based on such views our compiler framework dynamically generates the components of the implementing software system. Central to the compiler framework is the notion of generators, each generating a particular module for the application system and collectively implementing the intended application. Based on the resulting *modular architecture* the generated system allows personalized model definition and seamless model evolution.

In this paper we give details about the system modules as well as the generators which create them. We also describe how the generators are interconnected in the compiler framework.

The paper is organized as follows: in sec. 2 we give a brief account of conceptual modeling for content-intensive application systems and of contemporary approaches to model-based system generation. The additional support required for openness and dynamics is outlined in sec. 3. In sec. 4 we discuss our contributions to implementing open and dynamic content management systems. A detailed description of our model compiler framework is finally presented in sec. 5. The paper concludes with a summary and a short outlook on future work. Related work is discussed where appropriate.

2 Application System Modeling

Automatic system generation is based on abstract models—of either an application domain or of software. Such models and appropriate software generation facilities enable open dynamic content management systems.

2.1 Conceptual Modeling

Conceptual modeling [10, 8] is the activity of providing a model of an application domain. Conceptual modeling languages provide a domain vocabulary and avoid technical details as much as possible.

Starting system development with conceptual modeling thus avoids untimely consideration of technical constraints.

A conceptual model defines a vocabulary as a foundation for users and software developers. This way software uses the domain experts' vocabulary, and users are able understand the functionality of the developed software.

2.2 Model-driven Development

Research and practice in software engineering led to a thorough understanding of IS development. The insights gained are leading towards approaches which allow to derive software from specifications. To this end, models are used as (more or less) formal specifications of software systems.

Two of the approaches which are currently discussed take a somewhat exteme position: domain-specific languages (DSLs) [4] and mappings between software models expressed in general purpose languages, e.g., the Model-driven Architecture propsal [13].

DSLs are abstract languages for one application domain which are intended to be used by domain experts. DSLs are not necessarily (computationally) complete. Instead they cover an area of an application domain with a clearly defined scope. DSLs have a fixed semantics within the application domain. This semantics is based on by prefabricated software components which provide implementations. Such software components can range from libraries to software generators [17].

General purpose languages serve the modeling of complete software systems. Often, languages which allow varying degrees of concreteness are used, e.g. the Unified Modeling Language (UML). During the generation process of a software system a series of model mappings is applied, leading from abstract to more concrete models. During the process details are added at every model stage. Usually, the series of models starts with less formal models which can be created in cooperation with domain experts. Approaches based on general purpose languages generally arrive at completely formal descriptions of the software to be generated. Therefore, the final step of creating code comes down to a transformation from the chosen language to a programming language.

2.3 Open Modeling and Dynamic System Evolution

As discussed in the introductory section content management systems need

- a conceptual model which is open to different user views (openness) and
- implementations which keep up with the opinions of the users (dynamics).

Therefore, a modeling language is needed which on the one hand allows domain experts to describe their application domain, and on the other hand is concrete enough to serve the purpose of automatic software generation.

An approach which is purely based on a DSL does not offer openness since it is constrained to a fixed set of concepts which are offered by the respective language and mapped to existing software components. Existing approaches which map models expressed in a general purpose language to each other do not account for dynamics. The additional information given for the mapping at every model stage generally prevents fully automatic system creation [2, 5].

In contrast to the approaches discussed in the previous section we concentrate on the specific class of content management which is combined with conceptual models for the description of entities. Modeling is open to any application domain, while the restriction to systems with a common core functionality allows their dynamic generation.

Note that we have to consider two modeling facilities for openness and dynamics: for the *source* application domain we need a modeling language which is general enough to allow openness, meaning that it is not constrained to predefined concepts for the description of entities. The *target* software model has to be specific enough to allow dynamics by enabling automatic generation of content management systems.

3 Support for Openness and Dynamics

The requirements of openness and dynamics call for special support in both systems creation as well as in operation. In fact in open and dynamic systems the line between creation and operation is blurred.

3.1 Shortcomings of Manual Software Engineering

Data-intensive applications normally are developed in processes which bear resemblance with the waterfall process: As part of the analysis of an application domain a conceptual schema is created. Based on this schema, the whole of the system is implemented. This means that the application is manually linked to the schema by the implementation process. Obviously, any changes to the schema have an impact on all parts of the application.

Therefore, when openness and dynamics are required, the common approach of manually implementing a static conceptual schema is clearly unfeasible. Our approach can remedy the situation by open modeling and dynamic systems generation as is discussed in sec. 4. Model changes are considered the rule not the exception, and content management systems are created with evolution in mind.

3.2 Modeling Requirements

The conceptual modeling language is needed to mediate between two worlds: the application domain of a user and the later system implementation.

Our approach concentrates on content management but combines it with a conceptual model of the described entities. Entities are therefore modeled dualistically by medial content as well as a conceptual description (see sec. 4.1). The approach is thus applicable to a wide variety of application domains.

However, in order to support dynamic systems, the model given by the user has to be compiled into a running system without any human intervention. Our approach achieves this by means of a compiler framework running a set of generators as described in more detail in sec. 5. On top of the conceptual schema, some of the generators might require a few additional parameters to bridge the gap between a conceptual model and its implementation in a content management system. These parameters correspond to the additional information needed for general purpose language approaches as discussed in sec. 2.2.

3.3 System Requirements

The general requirements outlined above can be mapped to requirements to content management systems. In particular:

1. The conceptual schema needs to be available to users and users must be able to modify it.
2. The system must be up to date with any such modifications *automatically*, therefore any manual development is not possible.
3. The conceptual schema must be truly conceptual. In traditional systems development it is often the case that implementation decisions have to be made during the analysis phase for purely technical reasons (e.g., the length of fields because of restrictions in the database). Such information must be separated from the conceptual schema.

We describe in the next sections how these requirements are met by our approach.

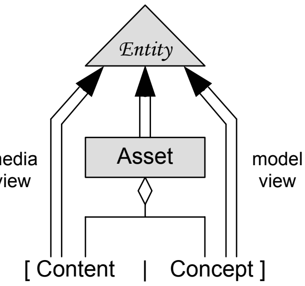


Fig. 1. Description capabilities of assets.

4 Ingredients of Open and Dynamic Systems

The requirements put forward in sec. 3 cannot be met with standard contemporary information systems. Generic systems lack openness since application domain concepts have to be mapped to generic ones, and hand-coded systems lack dynamics since changes require an incremental engineering roundtrip. Our approach employs a conceptual modeling language, a modular system architecture, as well as automatic system generation to meet the requirements. The conceptual modeling language is covered in detail in previous work (see [14, 15]) but will be outlined next. We then describe the modular architecture of generated content management systems and point out how these systems are created.

4.1 Conceptual Modeling Language

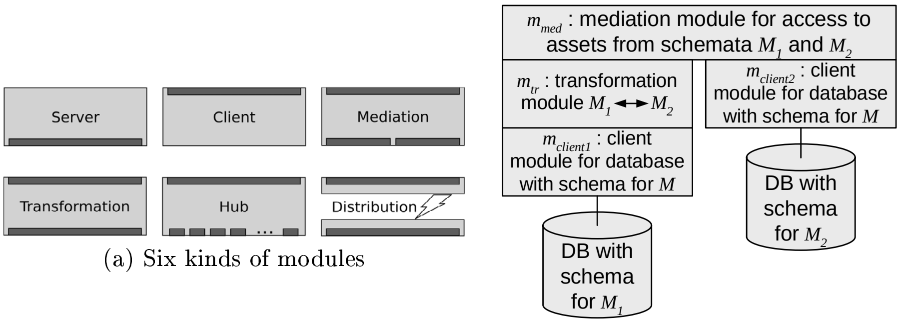
In our conceptual modeling language, entities are described by *asset classes*. These classes jointly describe a medial view (in the form of multimedia content, e.g., an image) and a conceptual view of the entity, see fig. 1. The conceptual view consists of characteristics (primitive attributes which are intrinsic to the entity), relationships to other asset classes, and constraints on the asset class. Classes are related in an inheritance hierarchy.

Asset classes are grouped into *asset models*, which usually deal with a particular application domain. Classes from one model can be imported into other models. The language supports openness by allowing the (partial) redefinition of imported classes to suite the task at hand. Anything that is not redefined stays the same as in the original.

Furthermore, the language provides means to create, modify, and delete instances as well as to query for them.

4.2 Modular System Architecture

The creation of a system in a dynamic manner can in some cases entail changes to its setup. The architecture of the system must therefore allow for flexible reconfiguration. A monolithic system is certainly not capable of such flexible



(a) Six kinds of modules

(b) A sample module organization

Fig. 2. Modules of generated content management systems.

change. Quite the contrary, we propose a modular system architecture that is built of many small modules. The kinds of modules for the most frequently occurring tasks are illustrated in fig. 2(a). All modules have a uniform interface and can be composed in layers. This makes it possible to always combine modules in the way most appropriate to the task at hand.

The module interface reflects the capabilities of the asset language to create, modify, delete and query for asset instances. Each module can thus express its functionality in terms of calls to the module(s) on the underlying layer.

A *component* is a combination of modules, usually arranged in layers. Components provide several services to their modules: resolution of identifiers, management of module lifecycles, and management of the proper organization of modules at system startup. Each module can use other modules and can also be used by several others. However, the setup of modules in a component always must be a directed acyclic graph.

Modules can be of several kinds, in particular:

– Components are accessed via *server modules* using standard protocols.
– The description data of asset instances is stored in third party systems, databases in most cases. Mapping asset models to schemata of such systems is done by *client modules*.
– A central building block of the architecture of generated content management systems is the mediator architecture [19]. In our approach it is implemented by modules of two kinds. The first are *mediation modules* which delegate requests to other modules based on the request.
– The other kind are *transformation modules*. By encapsulating mappings in such modules, rather than integrating this functionality into other modules, mappings can be added dynamically (compare [12]).
– *Hub modules* uniformly distribute calls to a larger number of underlying modules.
– By use of *distribution modules* components can reside at different physical locations and communicate by exchanging data.

These module kinds have been identified with respect to the requirements of content management systems. They provide basic services by the principle of Separation of Concerns. The functionality of a content management system is implemented by a *component configuration* which composes selected modules. For example, schema evolution leads to a combination of client, transformation, and mediation modules (indicated in figure 2(b), see [15] for details).

4.3 System Generation

The subdivision of a system into fine-grained modules as outlined above allows for flexible reconfiguration. This is necessary for a dynamic system, however not sufficient. Manual implementation of modules is unfeasible, as modules are usually highly schema dependent. System generation is therefore necessary to allow the system to be dynamic.

Several generative approaches (e.g., [18]) use loosely coupled generators. While this is fine for system generation under the supervision of a developer, generation in dynamic systems must happen without such intervention. We have therefore aimed at a tight, albeit flexible, coupling of generators. Given that our approach assembles systems of smaller modules, we can use generators which each create a particular type of module (e.g., a client module for persistence in relational databases). The generators are combined in a compiler framework which takes care of their proper setup and manages their interdependencies.

5 Model Compiler Framework

As argued in the previous section automatic software generation is necessary to allow dynamics of information systems with a fine-grained architecture.

There are different approaches to the problem of generating whole software systems which are composed of various parts that are produced by independent generators: (1) the generated software modules have to be adapted to be composable [7], (2) generic software modules are wrapped in a domain-specific way [11], (3) glue code to combine modules needs to be generated [3], or (4) the generators need to cooperate in order to create a consistent set of modules. As already indicated we favor the latter approach for content management systems.

Writing coordinated generators is a complex task, mainly because setting up an infrastructure for them [16] is difficult. Therefore, our model compiler for content management systems 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 [17].

5.1 A Framework Approach to Model Compilers

A typical compiler is divided into frontend and backend [1] to decouple source language recognition from target language generation. To this end, a compiler frontend creates an intermediate representation of the input definitions. Such an intermediate representation forms the input of a compiler's backend which generates code in the target language. This allows compiler setups for multiple targets as well as—at least in theory—to process different source languages.

The model compiler for our conceptual language is built in an object-oriented fashion. The classical division into frontend and backend has been translated into a framework architecture that allows to configure compilers for the generation of dynamic content management systems. This framework addresses the need to generate multiple targets in conjunction.

An instance of the compiler framework is defined by providing a parser, a dictionary proxy, several generators, and a configuration of the framework. This is detailed in the subsequent sections.

Alike a programming language compiler, which creates an intermediate code representation, the frontend in the compiler framework creates *intermediate model* representations. Starting from a class IntermediateModel the asset class definitions are available as an object graph.

Compilers use *symbol tables* to store information about the language constructs recognized. Our model compiler for content management systems builds on the concept of symbol tables, but extends it significantly: these tables are not only used in the frontend of a compiler, but they are the means by which generators communicate during the generation process.

Asset class definitions can be distributed: models are created by combining existing classes available to the modeler, and existing classes can be redefined (see section 4.1). Therefore, the model compiler needs access to asset classes which are not contained in the model at hand, but have been defined elsewhere. They are provided by *dictionaries* which store available class definitions.

Fig. 3 shows a UML sequence diagram of the frontend activities of a compiler run. This figure emphasizes the function of dictionaries. In the example an intermediate model im is created by the parser in the frontend. The definitions make reference to another model which is included as an intermediate model sm. In order to get access to it the parser requests it from the framework (sm=getModel). The framework contains a *dictionary proxy* to transparently access the known dictionaries. In the example there are two dictionaries. The first one does not know the requested model. The second one returns it as rm, and the dictionary proxy creates a local representation sm from rm.

Dictionary proxies are used to decouple a compiler configuration from the access to dictionaries. Dictionaries can be accessible by various means. For example, asset class definitions can be contained in local files or in resources accessible over a network. Dictionary proxies are a configurable part of the framework so that various alternatives can be realized.

Since a compiled model might be included into other models, it also has to be made available in a dictionary. In the example of fig. 3 the framework registers the model with the dictionary proxy (registerModel), which in turn inserts it into the first dictionary (createModel).

Dictionaries by themselves are content management systems which are generated from the asset meta model. This way, the compiler can use a proper

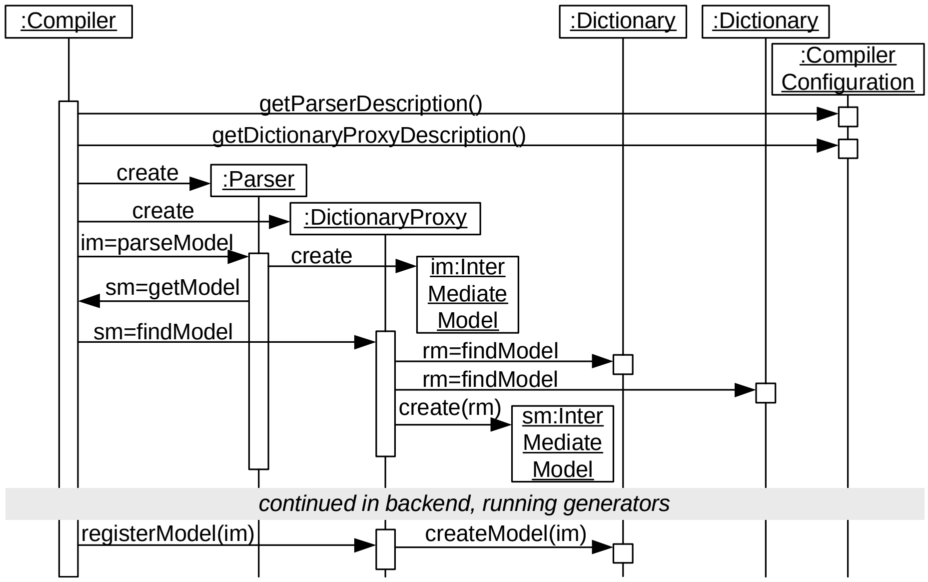


Fig. 3. Frontend of the model compiler with distributed dictionary.

component configuration. A variety of dictionary implementations can easily be created using the existing generators, e.g., dictionaries that store schemata in relational databases or XML databases. Furthermore, dictionaries can be equipped with a range of other functionality for, e.g., remote access.

5.2 Parsers

In accordance with the classical architecture of compilers the intermediate model is used to distribute information between the compiler components. An intermediate model is created from an external representation. A parser makes the compiler independent of a particular linguistic form of an asset model, and there are parsers that retrieve asset model definitions from various sources.

A set of parsers is readily available for model compiler instances. The one most commonly used reads files containing asset language expression as defined in [15]. Other options are parsers for different syntactical forms, e.g. in XML, or parsers that adapt an internal model representation from modeling tools.

Additional parsers can be developed within the compiler framework. They have to fulfill an interface prescribed by the framework which requires them to produce an IntermediateModel instance.

5.3 Code Generators

The backend of a model compiler consists of generators. There is a correspondence between generators and the modules of content management systems. For each implementation of one of the module kinds introduced in sec. 4.2 there is at least one generator. Often there is more than one generator which contributes to the creation of a module. For example, client modules for database access are typically created by a pair of generators; one of them creates the database schema, the other one creates code to access the database as well as to store and retrieve asset instances.

In order to be integrated into the compiler framework, generators have to fulfill an interface. This interface mainly defines methods for a protocol by which generators communicate with the framework.

As part of this protocol parameters can be passed to generators as will be explained in the following sections. Generators take parameter values into account when generating a module.

The artifacts which are created by a generator to implement a module are reflected in the symbol table of the generator. The generators create their artifacts as a complex structure into which the symbol table provides named points of entry. Each generator fills its symbol table during its execution and passes the symbol table back to the compiler framework afterwards. The framework in turn gives available symbol tables to further generators making them the essential means of generator communication.

The symbol tables contain detailed structured information about the artifacts which were created by the respective generator. A typical behavior for a generator is for example to iterate over all asset classes from the intermediate model and all their attributes to generate a piece of code for each attribute. The symbol table will then contain a mappings from attributes to these pieces of code (e.g., access methods). The aim of symbol tables is to make access explicit for generators which rely on artifacts created by others (and most generators do). Without symbol tables, generators further down the chain would have to make assumptions about namings and would have to recover the corresponding pieces from the whole of the generated artifacts.

A complete system is normally built from artifacts in several languages. Different meta-programming facilities are available to the generators to create their output. This facilitates the creation of structured models of the artifacts and is therefore important to provide meaningful symbol tables. The structured models are converted into their concrete form as a side-effect of the generator execution. Such a concrete form are for example files containing source code of a particular programming language.

5.4 Framework Configurability

By providing generator implementations the backend of the compiler framework is enriched with additional functionality. Which generators are actually executed is determined by a *compiler configuration*, as are the frontend components (parser, dictionary proxy) used. Multiple configurations can be provided by system experts. Upon dynamic system generation a user chooses one of the available configurations for each compiler run.

For the frontend, the parser and the dictionary proxy (see fig. 3) can be chosen. They are provided as discussed in sections 5.1 and 5.2.

The backend configuration consists of two kinds of definitions: the generators to be used for creating a content management system and values of parameter to the generators.

For each configuration a set of generator implementations is given. This way generator instances out of the known generator implementations are chosen. There may be more than one generator with the same implementation, for example, if two client modules for database access are needed in a content management system. In this case the two client modules are created by two instances of the corresponding generator. The generation results may differ because of different sets of parameter values.

Values for parameters which a generator might need are given as part of the configurations. Generators determine the parameters to use at runtime, and the framework will supply them with the values given in the configuration. This is part of the generator protocol introduced in the subsequent section.

5.5 Generator Control

Traditional compilers for programming languages include a backend for one generation target—executable binary code in most cases. In contrast a model compiler for content management systems has to consider several targets at a time, e.g., database schemata, database access code, application level code, and so on.

The multiple targets of a model compiler are addressed by the generators provided to the compiler framework as described in the previous section. The various artifacts a compiler creates are highly interrelated. Therefore, the execution of generators has to be scheduled in such a way that they create a working content management system.

Generators follow a specified protocol inside the compiler framework. Fig. 4 illustrates this protocol in the form of a UML sequence diagram. In this figure a compiler setup with three generators is shown. The APIGenerator is a standard generator which creates the uniform module interface (see sec. 4.2). An SQLSchemaGenerator produces a database schema for a relational database. Real setups use specialized generators which account for the peculiarities of concrete database management systems. The JDBCGenerator creates Java code for a client module which stores asset instances according to the given asset model in the database with the generated relational schema.

The grey box in fig.4 represents the compiler frontend as shown in fig. 3. It creates the intermediate model im which reflects the asset class definitions.

The extended symbol table concept described in sec. 5.1 is the primary means to coordinate generator executions. Depending on its configuration, the framework (here represented by the Compiler instance) creates the necessary generators. Each generator is asked for the symbol tables it needs as input, the symbol table it will produce as the result of a successful execution, and the configuration parameters it needs to be supplied with. Based on the information given by

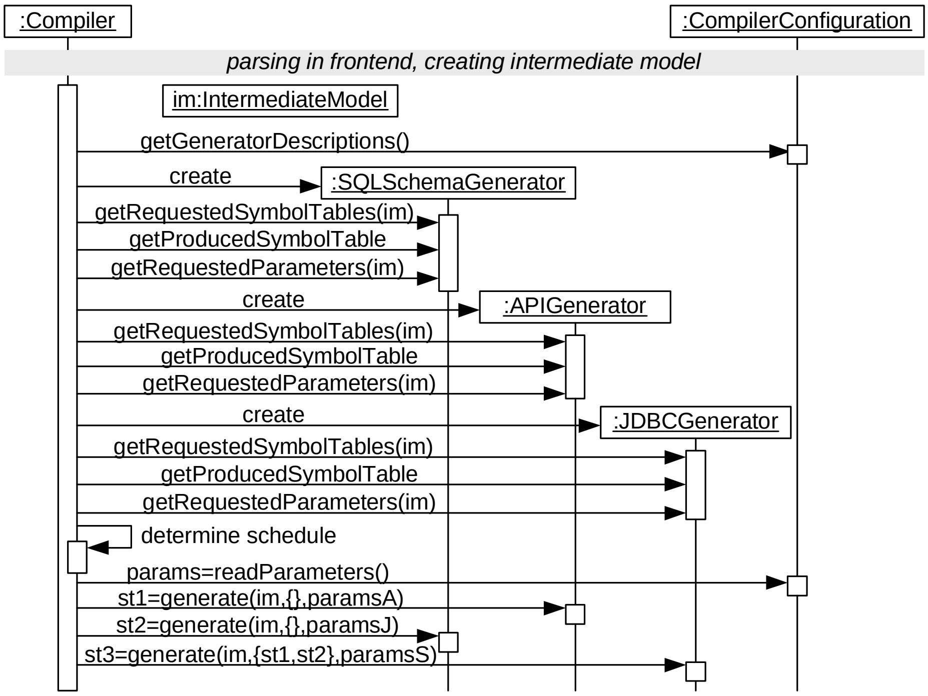


Fig. 4. Generator scheduling protocol.

the generators the framework computes a schedule for generator execution that ensures the required data flow.

In the example of fig. 4 both the API and the SQL schema generator will not require any symbol tables as input. The JDBC generator generates a client module which implements the module API and accesses a database configured with the generated schema, thus it requires symbol tables which reflect the respective artifacts. Therefore, the JDBC generator needs both symbol tables created by the other generators (st1 and st2) and thus has to be executed last. Either the API or the SQL schema generator can be run first, or both can be run in parallel. Following the generator protocol the JDBC generator returns the symbol table st3 as announced by getProducedSymbolTable. This symbol table can be used by generators which want to employ the JDBC code.

Finally, the generators are run in the determined order (generate in fig. 4). They are provided with the required symbol tables and parameter values, and return a new symbol table.

Fig. 5 makes the data flow that takes place between the generators through symbol tables more explicit. The generators of a first schedule stage (API and SQL schema generators) are executed concurrently. Each of them creates some module artifacts and stores information about the generated artifacts in a particular symbol table. The symbol tables are available to the generator of the second

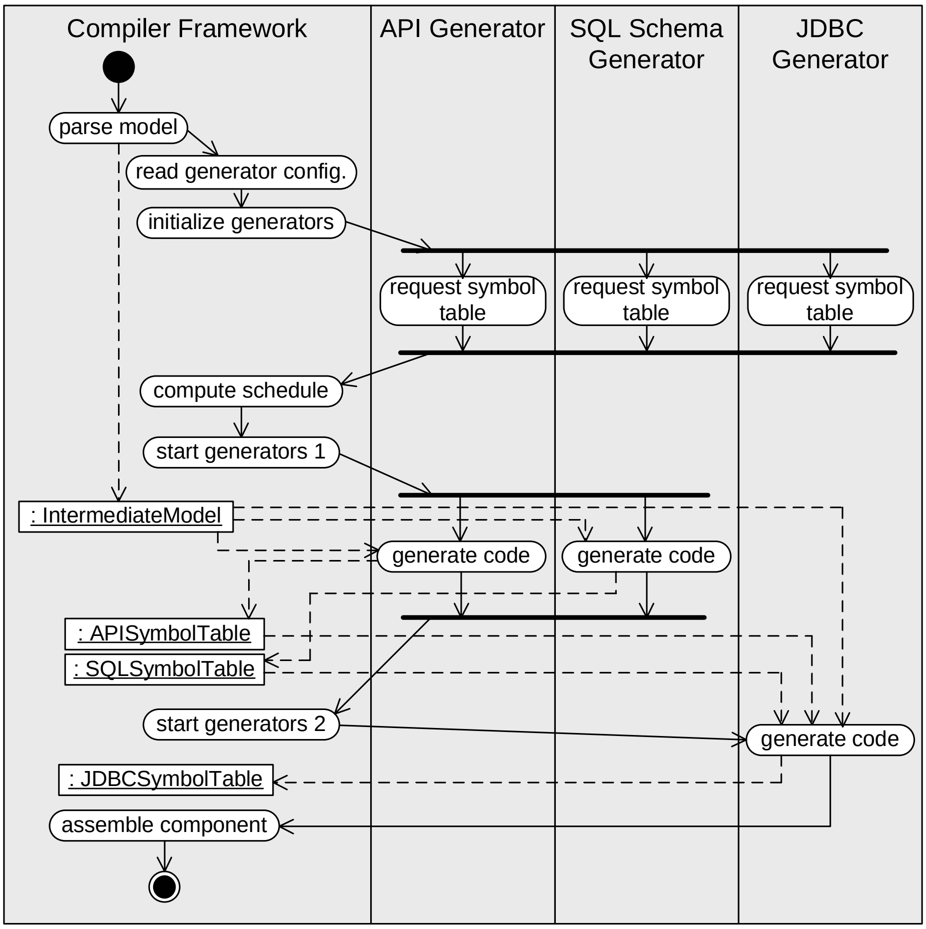


Fig. 5. Generator communication with symbol tables.

schedule stage (the JDBC generator). The activity diagram in fig. 5 shows both control and data flow to point out the fact that the compiler framework computes a schedule for the generators instead of having them controlled by data flow alone. This way, the compiler framework can detect inconsistent configurations without actually running generators.

Generators are provided with the intermediate model when they request symbol tables (getRequestedSymbolTables) and parameters (getRequestedParameters). This way the choice of symbol tables and parameters can depend on the actual asset class definitions. E.g., a schema generator needs type mappings for all asset class members. Therefore, it will gather the types used in asset class attributes and request the according SQL types which shall be used within a database. Because of the dynamic choice of symbol tables possible generator schedules can depend on the asset class definitions.

5.6 Component Assembly

When all generators have finished their tasks a system component is assembled from the generated modules and parameterizations of third party products. This includes two activities: actually building the modules and combining them in a component of a content management system.

Modules are built from the generated artifacts. Each generated artifact needs a special final treatment: source code needs to be compiled, database schemata have to be deployed, etc.

A component is created according to a given component configuration (see sec. 4.2) which determines the component's functionality on the basis of the basic functionality offered by the individual modules.

6 Summary and Outlook

Content management systems which describe domain entities by multimedia content have to take into account their users' views and working contexts. One way to do so is by means of a conceptual model provided by the users. In this paper we have presented a generative approach to the creation of content management systems that enables openness and dynamics of such systems.

As a solution to the problem of coordinating the various generators of software artifacts for a content management system a design for a compiler framework has been proposed.

Through application projects, we were able to verify that users are indeed enabled to provide their personalized perspective [9] and that a dynamic response to schema modifications is feasible [6].

In the future we hope to extend our approach in several respects. One of the focal points is a possible feedback of generator runs on the asset model. Giving such feedback will enable generators to interact with each other via the model to distribute any additional constraints that might be necessary to impose on the schema. An example of this is the length restriction of string fields. These restrictions can arise from the use of a relational database for persistence. However, these restrictions need to be respected by all parts of the application, e.g., in presentation logic. There is, therefore, a feedback loop from the client module generator providing additional information to the other generators. Currently such information which is important to all generators has to be defined in the asset model, violating its conceptual nature.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Scott W. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, third edition, 2004.

3. Uwe Assmann. Meta-programming Composers In Second-Generation Component Systems. In J. Bishop and N. Horspool, editors, *Systems Implementation 2000 – Working Conference IFIP WG 2.4*. Chapman and Hall, 1998.
4. Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153. IEEE, 1998.
5. Jorn Bettin. Model-Driven Software Development: An emerging paradigm for Industrialized Software Asset Development. Technical report, SoftMetaWare, 2004.
6. Sebastian Bossung, Hans-Werner Sehring, Patrick Hupe, and Joachim W. Schmidt. Open and Dynamic Schema Evolution in Content-intensive Web Applications. In José Cordeiro, Vitor Pedrosa, Bruno Encarnaçãom, and Joaquim Filipe, editors, *Proceedings of the 2nd International Conference on Web Information Systems and Technologies, WEBIST 2006*, pages 109–116. INSTICC Press, 2006.
7. Johan Brichau. *Integrative Composition of Program Generators*. PhD thesis, Vakgroep Informatica, Vrije Universiteit Brussel, 2005.
8. 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.
9. Matthias Bruhn. The Warburg Electronic Library in Hamburg: A Digital Index of Political Iconography. *Visual Resources*, XV:405–423, 2000.
10. Peter P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
11. Gopal Gupta. A language-centric approach to software engineering: Domain specific languages meet software components. In *Electronic Proceedings of the CoLogNet Area Workshop Series on Component-based Software Development and Implementation Technology for Computational Logic Systems*, Technical University of Madrid (Spain), 19.-20. September 2002.
12. Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology*. Kluwer, 2000.
13. Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical Report omg/2003-06-01, OMG, June 2003.
14. 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.
15. 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.
16. Yannis Smaragdakis and Don Batory. Scoping Constructs for Program Generators. Technical Report CS-TR-96-37, Austin, Texas, USA, 1996.
17. Yannis Smaragdakis, Shan Shan Huang, and David Zook. 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, 2004.
18. Pedro Valderas, Joan Fons, and Vicente Pelechano. Transforming Web Requirements into Navigational Models: An MDA Based Approach. In *Proc. ER05*, volume 3716 of *LNCS*, pages 320–336. Springer Verlag, 2005.
19. G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25:38–49, 1992.