Dynamic Metamodel Extension Modules to Support Adaptive Data Management

Michael Grossniklaus[1], Stefania Leone[2],
Alexandre de Spindler[2], and Moira C. Norrie[2]

[1] Dipartimento di Elettronica e Informazione, Politecnico di Milano
I-20133 Milano, Italy
grossniklaus@elet.polimi.it
[2] Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
{leone,despindler,norrie}@inf.ethz.ch

Abstract. Databases are now used in a wide variety of settings resulting in requirements which may differ substantially from one application to another, even to the point of conflict. Consequently, there is no database product that can support all forms of information systems ranging from enterprise applications to personal information systems running on mobile devices. Further, domains such as the Web have demonstrated the need to cope with rapidly evolving requirements. We define dynamic metamodel extension modules that support adaptive data management by evolving a system in the event of changing requirements and show how this technique was applied to cater for specific application settings.

1 Introduction

Nowadays, database systems support everything from enterprise applications to personal information management on mobile devices. These systems vary greatly, not only in terms of the scale, but also data management requirements, which may differ dramatically from one application to another. Currently, database products tend to dedicate themselves more or less explicitly to a given domain with major vendors offering either a family of products or means for design-time configurability. For example, the open-source database MySQL addresses a segment of the database market that differs substantially from that of products such as Oracle Database or IBM DB2, which were designed to support large-scale enterprise applications. In contrast, MySQL was originally developed to support Web applications and its kernel optimised for a subset of SQL to meet the requirements of that domain. MySQL also provides less support for transactional query processing since it is less relevant in typical Web applications.

In some cases, the requirements of one domain may even conflict with the requirements of another, making it impossible for one database product to support all domains. One reason for very different and even contradicting requirements is the fact that some of the features that are desirable during the design and development phase of a database application may be undesirable during operation.

B. Pernici (Ed.): CAiSE 2010, LNCS 6051, pp. 363–377, 2010.

For example, features such as schema evolution and versioning facilitate prototyping but often have adverse effects on system performance and may not be required for system operation. Moreover, many Web applications evolve rapidly and as a result the requirements may change dramatically during the operation.

The need for adaptation has already been recognised by the database community [1] and various proposals exist for *tailor-made data management* [2]. Traditional DBMS tended to be monolithic in structure and static in functionality. Over the last decades, there have been different approaches to allow a DBMS to be tailored to the requirements of particular applications and settings. The general goal was to find ways in which database systems could be made extensible and more configurable. While most of these approaches provide support for design-time adaptation, more recently approaches based on service-oriented architectures have been proposed that also support run-time adaptation.

Most existing approaches support adaptation through adaptable database architectures. In this paper, we propose a different approach, showing how adaptation in database systems can be supported by basing their implementation on a well-defined system metamodel that can be evolved through dynamic metamodel extension modules. Metamodel extension modules can be loaded and unloaded dynamically, thus uniformly providing database system adaptation at design-time and run-time. While our approach is generally applicable, we demonstrate it in the setting of an object database system.

We begin in Sect. 2 with a discussion of related work. Our approach together with the specification of metamodel extension modules is introduced in Sect. 3. The application of the proposed approach is demonstrated in Sect. 4 and 5 where the core metamodel and an example metamodel extension module are presented, respectively. Implementation details are presented in Sect. 6. Concluding remarks are given in Sect. 7.

2 Related Work

Adaptation of data management systems has been addressed in terms of lightweight, configurable database management architectures. The emergent trend of service-oriented architectures (SOA) has led to a number of proposals for service-oriented database architectures (SODA), e.g. [3,4], that allow application developers to configure a DBMS by linking together all services required for a given application domain. The general ideas for these architectures are very close to that of earlier configurable DBMS [5].

In [3], a service-oriented DBMS (SDBMS) architecture is described based on the layered architecture proposed in [6]. In addition to being able to extend functionality, it allows for the selection of an alternative service or service composition in the case of service failure. This could also involve introducing a wrapper to adapt the interface of a service. While the proposed architecture promises the flexibility required, there are no details about the mechanisms used to achieve the different kinds of flexibility and the implementation of the architecture. Further, it is not clear if and how run-time adaptation would be supported.

The CoBRA DB project [7] aims at providing run-time adaptation for DBMS. The focus lies on modularizing a DBMS and supporting module exchange at run-time in a transparent and atomic way. The authors experimented with two methods of enabling dynamic adaptation, namely dynamic aspect-oriented programming (d-AOP) and a second approach where a component implementation, or part of it, is exchanged in order to adapt the component while the interface remains valid. In [8], they present how a Transaction Manager can be added and removed at run-time using the d-APO approach which the authors argue in [7] has several disadvantages in terms of performance, code maintenance, limited functionality and testing. Therefore the approach of component replacement has been adopted in the CoRBA framework. In this approach, the adaptation manager is responsible for registering services offered by the components as well as for managing and triggering all adaptation requests.

In mobile applications, computing resources may be limited and it is therefore important that a DBMS can be configured and optimized for a particular application setting. COMET [9] is a component-based real-time database for automotive systems that represents a typical example of a DBMS that can be statically configured for a given application or target device. FAME-DBMS [10] is an approach to configurable DBMS in the area of embedded systems that follows the idea of software product lines with static system composition. DBMS functionality is tailored after the application has been developed to provide the minimal functionality required based on code analysis. For example, if the join operation is never used, the configured DBMS will not provide the operator. This approach is a design-time approach and run-time adaptation is not supported.

In summary, the ability to support various types of adaptation in database systems has been recognised as desirable, if not essential, in modern DBMS. Architectural solutions such as configurable DBMS are capable of supporting design-time adaptation, but they are not suited to run-time adaptation. While approaches based on SOA do support run-time adaptation, their motivation has mainly been to develop self-healing and self-managing systems in order to improve reliability. In contrast, our work focuses on supporting different and/or changing requirements of information systems brought about by wide variation in application settings and system evolution. We believe that in a setting where a DBMS is not distributed, for example applications on mobile devices, services are not ideally suited to solving the particular problem of adaptation since the fact that they are loosely coupled can have an adverse impact on system performance. Further, since many proposals adopt service-based approaches in which specific services are replaced or extended, it is difficult to support the types of adaptation that require adaptation across services such as storage, query processing and constraint management. For example, the kinds of adaptation that require changes to the basic structures of the data model to support spatial information or versioning may require changes to many parts of the system, especially if all data is to be handled uniformly. We therefore decided to investigate how a DBMS could be adapted through changes to the metamodel rather than to specific services.

3 Approach

The development of a database application typically involves defining a model of the application domain and implementing the means to create, retrieve, update and delete instances of the application domain concepts. The application model is itself defined in terms of the DBMS metamodel that specifies the core constructs supported by the system. In the case of relational technologies, the metamodel includes the concepts of relations and attributes, while object metamodels describe object types and their properties. Therefore, a DBMS must offer the basic database operations to create, retrieve, update and delete instances of the metamodel concepts in order to specify an application model. Also, most DBMS offer a database language (DBL) including data definition, data manipulation and data retrieval components.

Our approach assumes that data and metadata are handled uniformly, so that both the data model that defines the functionality of the DBMS and the application model are represented explicitly as data. This means that, not only may all database functionality such as storage management, query processing and constraint management be applied to metadata as well as data, but also they can be updated dynamically at run-time. The key to our approach to DBMS adaptation is to allow the core metamodel, corresponding database operations and the DBL to be changed or extended. This can be done either through a configuration process at design-time or by extending functionality dynamically at run-time to allow the DBMS to adapt to new requirements.

We claim that many requirements imposed by database applications can be met by extending the metamodel with additional concepts. Therefore, we decided to address the requirement of adaptive data management through a modular system metamodel. An overview of this approach is shown in Fig. 1.

The core database module $Module_{core}$ is shown on the left-hand side of Fig. 1. It comprises the core metamodel MM_{core} as well as the component to create, retrieve, update and delete core metamodel concepts ($CRUD_{core}$) and the core database language DBL_{core}. MM_{core} is a set of meta concepts $\{MC_1, ..., MC_n\}$.

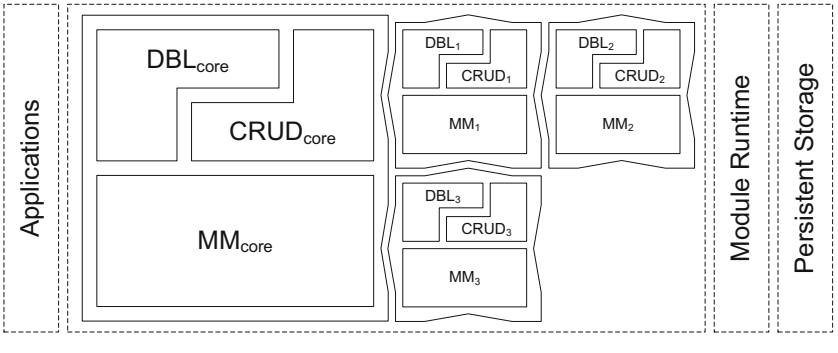

Fig. 1. Metamodel extension modules

As seen in Fig. 1, the definition of metamodel extension modules follows the design of the core system in the sense that each module also provides metamodel concepts, operations to manipulate them and an extension to the database language. The additional manipulation operations and database language extension for the new metamodel concepts are required to make the new functionality available to other parts of the system as well as to the application developer or end-user. In summary, database extensions consist of three components which, together, form what we refer to as a metamodel extension module.

More formally, a module can be defined as a triple

$$Module_{ext} = \langle MM_{ext}, CRUD_{ext}, DBL_{ext} \rangle$$

where MM_{ext} is a set of additional concepts made available to the application developer to model the application domain according to application-specific requirements, $CRUD_{ext}$ refers to the database operations and DBL_{ext} refers to the extensions to the database language.

In general, such an extension is a set of concepts $\{C_1, ..., C_m\}$ where each concept $C_i \in MM_{ext}$ is an instance of a meta concept $MC_j \in MM_{core}$, formally written as $C_i \triangleleft MC_j$. Note that, once the metamodel extensions have been added to the metamodel by loading the module, they become part of it and remain indistinguishable from the perspective of an application or developer. Therefore, the application developer can easily take advantage of these additional concepts in order to model the application domain.

The term $CRUD_{ext}$ refers to those facilities required for an application or developer to manage the instances of all concepts defined by the module.

$$\forall\ C_i\ \in MM_{ext}\ \exists\ CRUD_{ext}(C_i) \in CRUD_{ext}$$

where $CRUD_{ext}(C_i)$ allows for instances of the concept C_i to be created, retrieved, updated and deleted. When a module is loaded, the set of its operators $CRUD_{ext}$ are registered with the database in such a way that they can be retrieved and used by the developer or application. In a more general sense, the database operations can be seen as forming the API of the module.

The component DBL_{ext} is a set of symbols extending the core database language DBL_{core} to allow access to the operations offered by $CRUD_{ext}$. In general, a database language is defined by a grammar G consisting of a set N of nonterminal symbols, a set Σ of terminal symbols and the set P of production rules where each rule maps from one string of symbols to another. In short, the grammar can be written as $G = (N, \Sigma, P)$. Consequently, with each module loaded, the core language DBL_{core} defined by the core grammar G_{core} is extended by DBL_{ext} by unifying its grammar with the core grammar as

$$G_{core} \cup G_{ext} = (N_{core} \cup N_{ext}, \Sigma_{core} \cup \Sigma_{ext}, P_{core} \cup P_{ext}).$$

Moreover, there may exist dependencies among modules. By default, all modules are dependent on the core module. However, a module may additionally be dependent on other modules which means that they must be loaded first. Similarly, a module cannot be unloaded if other loaded modules depend on it. In order

for the module run-time to check dependencies, a list $[Module_1, \ldots, Module_n]$ of all dependent modules is defined as part of each module declaration.

Since the core components of our system also include a metamodel, database operations and a language, our system is built so that the core itself is defined as a module and loaded accordingly. In contrast to all other modules, the core module cannot be unloaded at run-time since all other modules depend on it. Nevertheless, the core module can be configured at design-time to adapt it, for example, to a mobile environment requiring lightweight databases or a heavily-used Web application relying on additional concepts to increase performance.

4 Core Metamodel

In this section, we present an example of a core database module consisting of a core metamodel MM_{core}, core management functionality $CRUD_{core}$ and a core database language DBL_{core}. Based on this, we will show the use of metamodel extension modules in the next section. As initially stated, our approach works independently of the given data model of a system as long as it is defined through a metamodel. Therefore, the approach can be equally applied to relational, XML and object databases. We have implemented the approach in the object database system OMS Avon [11] and therefore will present the details of the approach using this as an example.

We begin by introducing the concepts of the core metamodel MM_{core} shown in Fig. 2, which is based on the OM data model [12]. Essentially, the OM data model is an integration of entity-relationship (ER) and object-oriented models. In contrast to ER models where the concepts of entity types and entity sets are often merged, OM introduces a clear separation between the typing and classification of entities by using a two-level model. Each object has at least one object type that specifies the representation and behaviour of the object in terms of attributes and methods. Note that OM supports subtyping and also multiple instantiation which means that an object can be said to have multiple types. Objects are classified through membership in collections and each collection has a membertype that restricts membership to objects of a particular type. A collection is represented graphically as a shaded box with the membertype specified in the shaded part.

Just as types can be specialised through subtyping, classifications can be specialised through subcollections. A collection may have multiple subcollections and classification constraints such as *disjoint*, *cover*, *partition* and *intersect* may be placed over these. Note that for reasons of legibility, not all classification constraints are shown in Fig. 2.

Relationships in OM are represented by bi-directional associations that are defined in terms of a source and a target collection. Associations are a first-order concept of the model and are represented by binary collections. As in some extended ER models, cardinalities over associations are specified in terms of a minimum and maximum value that expresses the number of objects to which an object can be linked. Associations can also be specialised over collections. Associations are represented graphically as shaded ovals.

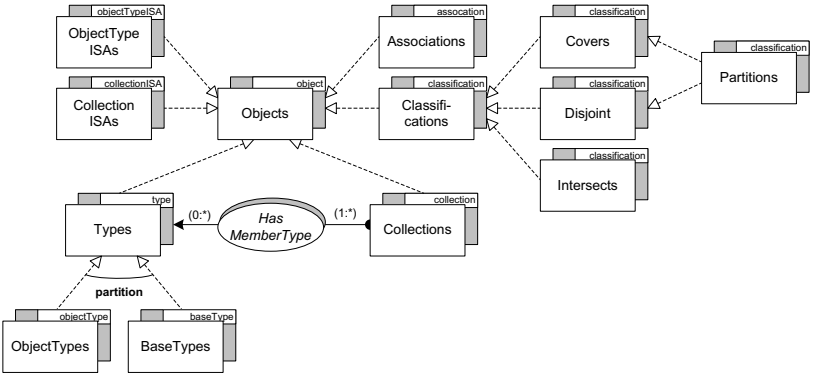

Fig. 2. Graphical representation of the core metamodel

ObjectTypes	Collections	Associations
create(Name): ObjectType retrieve(Name): ObjectType getName(ObjectType): Name addAttribute(ObjectType, Name, Type) remove Attribute(ObjectType, Name) getAttributes(ObjectType): Collection delete(ObjectType)	create(Name, MemberType): Collection retrieve(Name): Collection getName(Collection): Name getMemberType(Collection): MemberType addMember(Collection, Member) removeMember(Collection, Member) delete(Collection)	create(Name, Domain, Range, Relation, ...): Association retrieve(Name): Association getName(Association): Name getDomain(Association): Collection getRange(Association): Collection getRelation(Association): Collection addMember(Association, Member, Member) removeMember(Association, Member, Member) delete(Association)

Fig. 3. UML class definitions of the core system operators

As can be seen in Fig. 2, the instances of all constructs of the core model—types, collections, associations, ISA relationships (subtypes and subcollections) and also classification constraints—are represented as objects. These objects are classified through membership of the corresponding metadata collections.

Formally, the concepts defined by the core metamodel MM_{core} are given by its set of types, its set of collections and its set of associations.

$$\begin{aligned} MM_{core} = \{&\{object, type, collection, association, \ldots\}, \\ &\{Objects, Types, Collections, Associations, \ldots\}, \\ &\{HasMembertype, \ldots\}\} \end{aligned}$$

It is beyond the scope of this paper to describe all aspects of the OM model in detail.

In Fig. 3, we show UML type definitions of the manipulation operators for these concepts. These diagrams show the methods to create, retrieve, update and delete instances of the corresponding type. For example, the creation of a collection takes the name and membertype of the collection as argument, internally creates an object, dresses it with the collection type, sets the name and membertype attributes and returns this object. Given a collection object, its name and membertype can be retrieved using the getName and getMemberType

methods. An object dressed with this membertype can be added to or removed from the collection using the addMember and removeMember methods. Finally, the delete method is used to delete a collection object.

The third part of the core module is the database language DBL_{core}. Associated with the OM data model, we have defined the OML language [13] which encompasses a data definition, data manipulation and query language. The query language is based on a collection algebra that defines a set of operators to manipulate and process collections and associations. Apart from being used for data definition, manipulation and querying, OML also serves as a declarative object-oriented implementation language for the methods of database objects as well as for stored procedures and triggers. An example of an OML script is given below.

```
/* data definition language */
create type contact ( name : string, phone : string );
create collection Contacts as set of contact;
/* data manipulation language */
$obj := create object;
dress $obj with contact ( name = "Fred Bloggs", phone = "555-2223344" );
insert [ $obj ] into Contacts;
/* query language */
$fred := first(all $c in Contacts having ($c.name like "(F|f)red.*"));
```

In the data definition section, the application developer creates an object type contact, which is used as membertype for collection Contacts. The first statement creates an object of type objectType in the core metamodel, while the second statement creates a collection object. The data manipulation section demonstrates how an object is created and instantiated with the contact type using the `dress` operation. Then the object is inserted into the Contacts collections. Finally, a simple selection query over the Contacts collection is shown that selects the previously created object. Formally, OML is defined by a grammar expressed as a set of productions P_{core}. For reasons of space, only a subset of P_{core} is given below.

$$
\begin{aligned}
\text{statements} &\rightarrow \text{statement } \{ \texttt{";"} \text{ statement } \} \\
\text{statement} &\rightarrow [\ \text{ddl\_statement} \mid \text{dml\_statement} \mid \text{query\_expression}\ ] \\
\text{ddl\_statement} &\rightarrow \text{create\_statement} \\
\text{create\_statement} &\rightarrow \texttt{"create"}\ [\ \text{create\_object} \mid \text{create\_objecttype} \mid \ldots\ ] \\
\text{create\_object} &\rightarrow \texttt{"object"} \\
\text{create\_objecttype} &\rightarrow \texttt{"type"}\ \textit{name}\ \texttt{"("}\ \text{attribute\_list}\ \texttt{")"} \\
&\ldots
\end{aligned}
$$

Correspondingly, the DBL_{core} component is given by

$$
\begin{aligned}
DBL_{core} = \{ &\{\text{statements}, \text{statement}, \text{ddl\_statement}, \ldots\}, \\
&\{\texttt{"create"}, \texttt{"object"}, \texttt{"type"}, \ldots\}, P_{core}\}.
\end{aligned}
$$

5 Metamodel Extension Module

In this section, we show how we produced an object database with integrated support for Web content management by defining the appropriate metamodel extension module based on previous work [14] that established four information concepts for content management: content, view, structure and layout. Content elements represent the objects that are to be published on the Web, view elements define which attributes and relationships of these objects are displayed, structure elements provide support for arbitrarily nested content hierarchies, and layout elements govern the presentation of content and structure. Due to space limitations, we will omit further discussion of view elements here. As all of these elements are context-aware, the resulting system is very flexible and well-suited to support personalisable and multi-channel Web applications. Context-awareness is supported based on a version model [15] that manages each object as a set of variants which are defined for specific context states. At runtime, the client context state is matched against the variant context states and the best matching variant is selected to represent the object.

The metamodel of the extension module is shown in Fig. 4. On the left, the hierarchy of content management concepts is shown. On the right, the object variants for context-awareness are shown. Note that we support object variants based on multiple instantiation similar to the approach presented in [16]. To publish an existing data object on the Web, it is simply dressed with an instance of type variant that augments the object with context state information. Then it is associated with an instance of type content that assigns a resource name to the content for referencing it based on a URL. Elements and variants are linked with two associations. HasVariants captures all context variants of an element, while DefaultVariant designates a fallback representation of the object that can be used, for example, in the absence of a client context state. In order to make the application of layout elements to content elements type-safe, both concepts are associated with a type. Since Types is a concept of the core metamodel, the content management metamodel is an extension of the core.

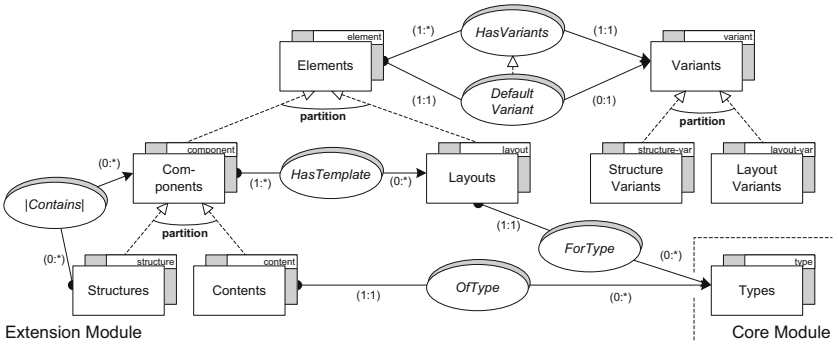

Fig. 4. Graphical representation of the content management metamodel

To implement this system as a module, the three module components have been defined as follows. The metamodel is a definition of the types, collections and associations introduced above.

$$MM_{cm} = \{\{element, variant, component, layout, structure, \ldots\},\\ \{Elements, Variants, Components, Layouts, Structures, \ldots\},\\ \{HasVariants, DefaultVariant, HasTemplate, \ldots\}\}$$

Note that the collections Elements and Variants are subcollections of the Objects collection (not shown in the figure) of the core metamodel. The system operators providing the creation, retrieval, update and deletion of these metamodel concepts are defined as

$$CRUD_{cm} = \{Content, Structures, Layouts, Variants\}.$$

Those operators relevant to the client of the content management system are shown in Fig. 5. Note that the operators shown implement just the management operations required in order to interact with the content management system. The part of the system that publishes the content on the Web is outside the scope of this paper. The presented operations make use of the core operators in $CRUD_{core}$ in order to implement their functionality. For example, the create operation in Variants calls the create operation in Objects, takes the returned object and dresses it with the variant type before returning it.

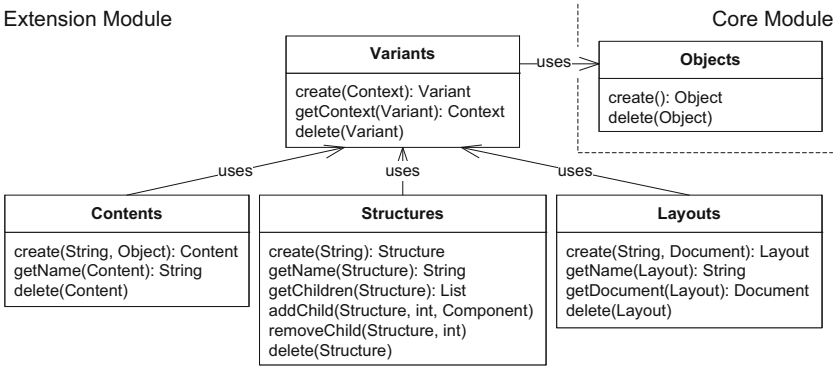

Fig. 5. UML class definitions of the content management system operators

Finally, the database language extension DBL_{cm} provides the vocabulary allowing the operations offered by $CRUD_{cm}$ to be invoked. The example below shows how to setup the content management to publish a created object `$obj` on the Web.

```
/* Create content element*/
create content fred from $obj context ( lang = "english" ) default;
/* Create main structure */
create structure index context ( lang = "english" ) default;
insert [ fred ] into index;
/* Create a template for person content */
create layout contact_layout for content context ( lang = "english" )
   [ <xsl:template match="..."> ... </xsl:template> ];
```

The extensions to the core language as defined in the previous section are formally defined by the set of productions P_{cm}. Again, we limit ourselves to a subset due to space limitations.

$$
\begin{aligned}
\text{create\_statement} &\rightarrow \texttt{"create"}\ [\ \ldots\ |\ \text{create\_content}\ |\ \text{create\_structure}\ |\ \text{create\_layout}\ ]\\
\text{create\_content} &\rightarrow \texttt{"content"}\ \textit{name}\ \texttt{"for"}\ \text{object\_ref}\ [\ \text{context}\ ]\\
\text{create\_structure} &\rightarrow \texttt{"structure"}\ \textit{name}\ [\ \text{context}\ ]\\
\text{create\_layout} &\rightarrow \texttt{"layout"}\ \textit{name}\ \texttt{"for"}\ \text{type\_ref}\ [\ \text{context}\ ]\ \texttt{"["}\ \text{template}\ \texttt{"]"}\\
\text{context} &\rightarrow \texttt{"context"}\ \text{value\_list}\ [\ \texttt{"default"}\ ]\\
\text{insert\_statement} &\rightarrow \texttt{"insert"}\ [\ \text{collection\_insert}\ |\ \text{structure\_insert}\ ]\\
\text{structure\_insert} &\rightarrow \text{component\_ref}\ \texttt{"into"}\ \text{structure\_ref}\\
&\ldots
\end{aligned}
$$

These productions lead to the definition of the DBL_{cm} component as

$$
\begin{aligned}
DBL_{cm} = \{&\{\text{create\_content}, \text{create\_structure}, \text{create\_layout}, \text{context}, \ldots\},\\
&\{\texttt{"content"}, \texttt{"structure"}, \texttt{"layout"}, \texttt{"context"}, \ldots\}, P_{cm}\}.
\end{aligned}
$$

To conclude this section, we note that the approach has also been used to develop an object database that allowed personal information to be integrated with Web 2.0 data sources, to extend an object database to support event-based programming [17] and to develop a platform for peer-to-peer data sharing in mobile applications [18].

6 Implementation

As shown in Fig. 1, the module runtime of our adaptive database system is built on top of a low-level persistent storage designed to provide flexible data management. The flexibility is achieved by means of the data model outlined in Fig. 6. We distinguish the notion of an *object* which strictly identifies a real-world object and an *instance* which bears the attribute values declared by an *object type*. An *extent* is a bulk of values that are described by an *extent type* and used to support collections and associations. Note that attribute values and extent members may be objects, extents or built-in values such as integer or string.

The persistent storage implements persistent data management according to this data model and exposes the API shown in Fig. 7. Object types are created with a list of attribute definitions, each declaring the name and type of an attribute. An extent type is created by providing the membertype. In both cases, an object must be provided which will serve as an identifier referring to the created type.

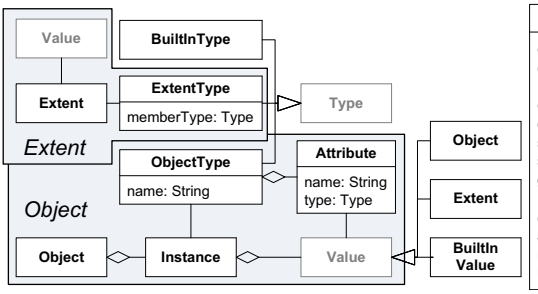

Fig. 6. Data model of persistent storage

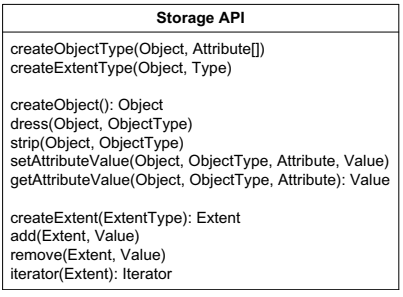

Fig. 7. Persistent storage API

The API offers methods to manage objects and extents. Once an object has been created, instances can be added or removed using the `dress` and `strip` methods, respectively. Attribute values can be set and retrieved by providing the object, the object type declaring the attribute to be accessed and the attribute itself. An extent is created by providing the extent type. Given an extent, values can be added and removed as well as accessed by means of an iterator. Methods for the deletion of types, objects and extents as well as a query facility are also provided, but not shown in the figure.

The module runtime provides a single Java class `OMObject` that represents all metamodel concepts uniformly in terms of the notion of an object as defined by the persistent storage data model. It offers means to add and remove instances as well as to access and manipulate attribute values. Another class `OMExtent` wraps the notion of an extent and allows for members to be added, accessed and removed. By separating the metamodel concepts from the actual concept representation within the programming language, we achieve the flexibility of being able to alter and extend the metamodel at runtime. Therefore, altering and extending the metamodel in the database does not require any changes to the in-memory Java representations of metamodel concepts.

To manage modules, the module runtime provides a `ModuleManager` that offers methods to load and unload modules. When the manager is initialised, it reads a configuration file where modules may be specified at design-time. The module manager requires that a module implementation follows the `Module` interface that is also defined by the runtime. This interface defines four methods that correspond to the lifecycle of modules. To load a module, the module concepts are created by the `bootstrap` method, then the operators are initialised with `registerCRUDs` and, finally, `generateDBL` loads the database language. If no longer required, the manager disposes of modules by invoking the `unload` method.

In the remainder of this section, we discuss the implementation of the core and the extension modules presented in Sects. 4 and 5, respectively. Due to the fact that all extension modules are implemented using the core module, the implementation of the core module differs from that of an extension module. For both modules, we will present an excerpt of the bootstrap and the operator implementation. To illustrate the discussion, Fig. 8 gives a UML interaction diagram that shows the communication between all involved actors.

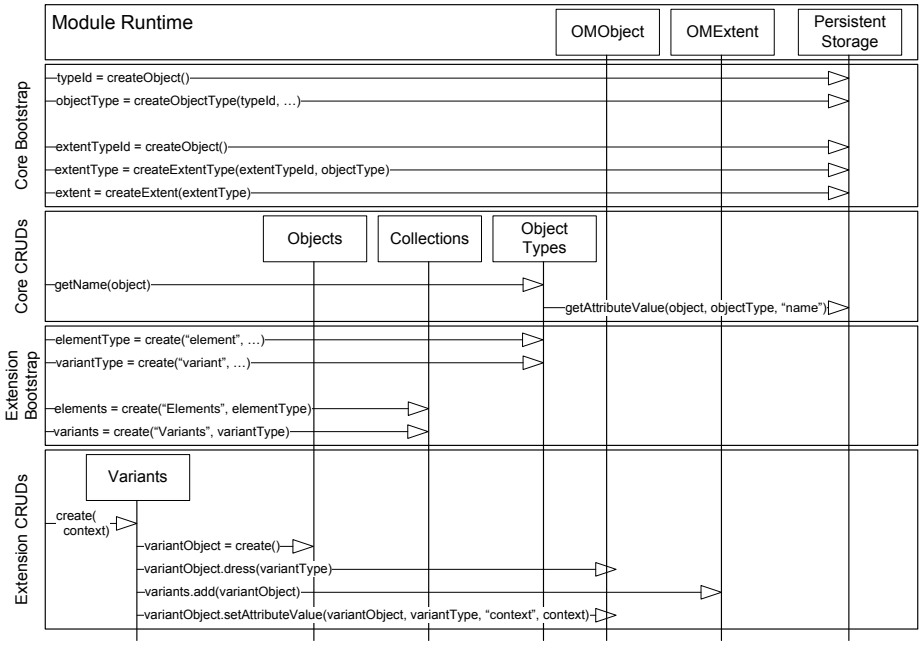

Fig. 8. UML interaction diagram of core and extension bootstrap and operators

The core metamodel is implemented entirely using the persistent storage shown in Fig. 7. The bootstrap process creates the core concepts represented in terms of types and collections and their relationships using the methods exposed by the persistent storage. As shown at the top of Fig. 8, the core database module creates the type objectType using the `createObjectType` method. The collection ObjectTypes is created by first creating the extent type using the method `createExtentType` and then creating the extent using `createExtent`. All other types and collections are created similarly.

Below the excerpt of the core bootstrap, Fig. 8 shows how the core operators make use of the persistent storage API to implement their concept-specific services. They take an `OMObject` instance as argument—or return one—and encapsulate the creation, deletion, attribute access and manipulation implemented using the methods of the persistent storage. For example, the object type operator to create an object type gathers the required arguments and calls the `createObjectType` method. The read and manipulation operations make use of the `setAttributeValue` and `getAttributeValue` methods. In the figure, we give an example of the latter, showing how the `getName` method of `ObjectTypes` uses the storage to retrieve that name of an object type. Finally, retrieval is implemented using the querying facility.

Extension modules are completely decoupled from the persistent storage as they only interact with the module runtime and the core module. The extension bootstrap procedure shown near the bottom of Fig. 8 consists of using the

core operators to create instances of the core metamodel representing extension concepts. For example, the content management module uses the core operator `ObjectTypes` to extend the metamodel with object types such as element and variant. In the same way, it uses the `Collections` operator to create the corresponding collections, i.e. Elements and Variants.

Finally, at the bottom of Fig. 8, we present how the extension operators use the core operators to provide the creation, attribute access and manipulation, retrieval and deletion facilities of the extension concepts. We use the example of the `Variants` operator to show how its `create` method has been implemented based on the core operator `Objects` as well as the module runtime classes `OMObject` and `OMExtent`.

To generate the database language of a module, we currently use JavaCC[1] and, therefore, the grammar is expressed in terms of the JavaCC syntax. Each module provides its grammar as a single file that is merged with the corresponding grammar files of other modules to obtain a comprehensive grammar before generating the parser. As a consequence, we can only support design-time adaptation for modules that require a language extension since parser generation requires an additional compilation step. To the best of our knowledge, there are no compiler compilers available at the moment that overcome this limitation and we are investigating how to engineer a solution that also provides runtime adaption of the database language.

7 Conclusions

We have motivated the need for adaptive database management systems to support configuration at design-time and evolution at run-time and proposed an approach that is based on revisions to the system metamodel. We have shown how this can be implemented and also demonstrated the use of the approach by means of an example. Even though the paper describes the application of the approach in the setting of an object database, it is important to emphasise that the approach generalises to all systems that use well-defined metadata to describe the data that they manage. While our main focus to date has been on achieving the desired functionality, we recognise that such flexibility of adaptation comes at a price in terms of performance and are now investigating exactly what the overhead is and how it can be reduced.

References

1. Stonebraker, M., Cetintemel, U.: "One Size Fits All": An Idea Whose Time Has Come and Gone. In: Proc. Intl. Conf. on Data Engineering, ICDE (2005)
2. Apel, S., Rosenmüller, M., Saake, G., Spinczyk, O. (eds.): Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management. University of Magdeburg (2008)

[1] `http://javacc.dev.java.net/`

3. Subasu, I.E., Ziegler, P., Dittrich, K.R., Gall, H.: Architectural Concerns for Flexible Data Management. In: Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management, SETMDM (2008)
4. Tok, W.H., Bressan, S.: DBNet: A Service-Oriented Database Architecture. In: Proc. Intl. Conf. on Database and Expert Systems Applications, DEXA (2006)
5. Dittrich, K.R., Geppert, A. (eds.): Component Database Systems. Morgan Kaufmann, San Francisco (2001)
6. Härder, T.: DBMS Architecture – New Challenges Ahead. Datenbank-Spektrum 14 (2005)
7. Irmert, F., Fischer, T., Meyer-Wegener, K.: Runtime Adaptation in a Service-Oriented Component Model. In: Proc. Intl. Workshop on Software Engineering for Adaptive and Self-managing Systems, SEAMS (2008)
8. Irmert, F., Lauterwald, F., Neumann, C.P., Daum, M., Lenz, R., Meyer-Wegener, K.: Semantics of a Runtime Adaptable Transaction Manager. In: Proc. Intl. Database Engineering & Applications Symposium, IDEAS 2009 (2009)
9. Nyström, D., Nolin, M., Norström, C., Hansson, J.: COMET: A Component-Based Real-Time Database for Automotive Systems. In: Proc. Workshop on Software Engineering for Automotive Systems (2003)
10. Rosenmüller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, S., Leich, T., Spinczyk, O., Saake, G.: FAME-DBMS: Tailor-Made Data Management Solutions for Embedded Systems. In: Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management, SETMDM (2008)
11. Norrie, M.C., Grossniklaus, M., Decurtins, C., de Spindler, A., Vancea, A., Leone, S.: Semantic Data Management for db4o. In: Proc. Intl. Conf. on Object Databases, ICOODB (2009)
12. Norrie, M.C.: An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In: Elmasri, R.A., Kouramajian, V., Thalheim, B. (eds.) ER 1993. LNCS, vol. 823. Springer, Heidelberg (1994)
13. Lombardoni, A.: Towards a Universal Information Platform: An Object-Oriented, Multi-User, Information Store. PhD thesis, ETH Zurich, Zurich, Switzerland (2006)
14. Grossniklaus, M., Norrie, M.C.: Information Concepts for Content Management. In: Proc. Intl. Workshop on Data Semantics in Web Information Systems (2002)
15. Grossniklaus, M., Norrie, M.C.: An Object-Oriented Version Model for Context-Aware Data Management. In: Benatallah, B., Casati, F., Georgakopoulos, D., Bartolini, C., Sadiq, W., Godart, C. (eds.) WISE 2007. LNCS, vol. 4831, pp. 398–409. Springer, Heidelberg (2007)
16. Beech, D., Mahbod, B.: Generalized Version Control in an Object-Oriented Database. In: Proc. Intl. Conf. on Data Engineering (1988)
17. Grossniklaus, M., Leone, S., de Spindler, A., Norrie, M.C.: Unified Event Model for Object Databases. In: Proc. Intl. Conf. on Object Databases, ICOODB (2009)
18. de Spindler, A., Grossniklaus, M., Norrie, M.C.: Development Framework for Mobile Social Applications. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 275–289. Springer, Heidelberg (2009)