Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

A Platform to Support Object Database Research

Michael Grossniklaus[a] Stefania Leone[b] Alexandre de Spindler[c]
Moira C. Norrie[d]

a. Department of Computer and Information Science
University of Konstanz, 78457 Konstanz, Germany

b. Department of Computer Science
University of Southern California, Los Angeles, CA 90089, USA

c. Centre for Business Information Management
Zurich University of Applied Sciences, 8400 Winterthur, Switzerland

d. Department of Computer Science
ETH Zurich, 8092 Zurich, Switzerland

Abstract Databases play a key role in an increasingly diverse range of applications and settings. New requirements are continually emerging and may differ substantially from one domain to another, sometimes even to the point of conflict. To address these challenges, database systems are evolving to cater for new application domains. Yet little attention has been given to the process of researching and developing database concepts in response to new requirements. We present a platform designed to support database research in terms of experimentation with different aspects of database systems ranging from the data model to the distribution architecture. Our platform is based on the notion of metamodel extension modules, inspired by proposals for adaptive and configurable database management systems. However, rather than building a tailored system from existing components, we focus on the process of designing new components. To qualitatively evaluate our platform, we present a series of case studies where our approach was used successfully to experiment with concepts designed to support a variety of novel application domains.

Keywords Object databases; Metamodel extension modules; Research platform.

1 Introduction

A wide variety of applications, ranging from enterprise and automated business systems to personal information management on the desktop or mobile devices, require the

Michael Grossniklaus, Stefania Leone, Alexandre de Spindler, Moira C. Norrie. *A Platform to Support Object Database Research.* In Journal of Object Technology, vol. 12, 2013, pages 2:1–31.
doi:10.5381/jot.2013.12.2.a2

use of database management systems (DBMS). Emerging application domains often give rise to new research challenges and, ultimately, novel database technologies. For example, user-generated content in Web 2.0 introduced new requirements for managing and sharing personal data, while the emergence of mobile social applications has led to opportunistic data sharing and location-based data management.

Even though the database community has acknowledged that catering for all application domains in a single system is difficult or even impossible [SC05], little attention has been paid to how the research and development of novel database technologies can be better supported. We believe there is a need for platforms that support research in terms of design, prototyping, testing and deployment of such technologies in order to quickly respond to new requirements.

The features of such a platform to support object database research go beyond the current forms of tailor-made data management [ARSS08] supported by adaptable and configurable database systems. Most of these approaches are based on a general system architecture that predefines components and interfaces. Adaptation to a specific application or use case is achieved by configuration, i.e. default components can be replaced with custom implementations or excluded if not required. In the setting of database research, this architectural approach suffers from the limitation that it focuses on how *existing* components are implemented rather than how *new* components can be supported. It is therefore difficult to extend the underlying data model with novel concepts, without replacing major parts of the system.

We present an approach that focuses on structuring the development of new database components, rather than building tailored systems from existing components. Consequently, database adaptation and extension in our approach is defined at the conceptual, rather than technological level. At its core, our approach is based on a well-defined system metamodel with adaptations and extensions of the database realised by evolving the system metamodel using so-called *metamodel extension modules*. In comparison to the architectural approach, the common system metamodel shared by all modules replaces the need for a standardised architecture with predefined database components. As we will see, the common metamodel is also the basis for specifying elaborate contracts between modules to support more advanced forms of refinement and reuse than is possible in the architectural approach.

Our metamodel extension modules consist of three parts: a metamodel extension defining new concepts, a library providing functionality to manage the new concepts, and a language extension to interact with the new concepts. This design is motivated by the observation that the requirements of novel application domains can affect the database at the data storage, functionality and/or interface level. To support the prototyping and testing of novel database technologies, metamodel extension modules can be loaded and unloaded dynamically. While our approach is generally applicable since most databases define a system metamodel at their core, we demonstrate it in the setting of an object database system. Object databases are typically tightly integrated with object-oriented programming languages and therefore requirements at the interface level are more pronounced than with other types of database systems. Accordingly, we have chosen to present typical case studies that reflect use cases of object databases in a mostly single-user and embedded setting.

This article is an extended presentation of Grossniklaus *et al.* [GLdN10], where we first introduced the notion of metamodel extension modules. While we previously focused on metamodel extension modules as a mechanism to support adaptation in databases, the present article discusses how this module mechanism enabled us to

build a platform to research object database technologies. Additionally, this platform is qualitatively evaluated based on the presentation of a series of case studies that demonstrate how we used it to prototype solutions for emerging database application domains. As mentioned above, new requirements can affect a database system in different ways and we have carefully selected the case studies to illustrate how our module mechanism copes with various types of requirements. To summarise, the main contributions of this article are as follows.

- A platform to support object database research must meet certain requirements. In this article, we precisely define these **requirements** and outline the **challenges** of building such a platform.
- We present the realization of our platform to support object database research in terms of a **formal approach**, so-called metamodel extension modules, and a detailed presentation of its **implementation**.
- A series of **case studies** and a **critical discussion** provide a qualitative evaluation of our approach in terms of accomplishments and open issues.

The remainder of this article is structured as follows. We begin in Section 2 with a discussion of the state of the art and the background in which our work is situated. We then motivate the challenges of building a platform to support object database research in Section 3. Section 4 formally introduces the concept of metamodel extension modules that is the enabling technology for this platform. In Section 5, we present the design, implementation and use of the platform. Section 6 discusses a series of case studies that give qualitative evidence as to the validity of our approach. In Section 7, we discuss and evaluate our approach with respect to the challenges outlined at the beginning of the paper. Finally, concluding remarks are given in Section 8.

2 Background

Current database products either tend to focus more or less explicitly on a single domain or only provide limited adaptation capabilities. To support adaptation, the monolithic structure and static functionality of a traditional DBMS is replaced with some form of component-based architecture. Generally, a Component DBMS (CDBMS) is composed of a set of core components and can be extended by introducing new components for an application domain [DG01]. A component typically offers a well-defined set of functionalities such as an algorithm or the implementation of a specific data type and application developers configure a DBMS by linking components together.

CDBMS can be classified into four categories [DG01], ranging from rather monolithic approaches with well-defined extension points to architectures where the DBMS can be configured in many different ways. A *pluggable* DBMS architecture provides well-defined extension points, where components with specific functionality such as abstract data types (ADT) or special indexes can be plugged into a DBMS that is based on System-R [SMA+07]. Examples of commercial System-R based systems with support for ADT are Oracle Database[1] and DB2[2]. Another category is *database middleware*, which integrates existing components at the data store level using an integration layer. In the case of *DBMS services*, database functionality is bundled into

[1] `http://www.oracle.com/database/`
[2] `http://www.ibm.com/software/data/db2/`

loosely-coupled services. Finally, *configurable* DBMS architectures extend the idea of DBMS services by supporting service implementations, which can be exchanged or adapted to new requirements.

The emergent trend of service-oriented architectures (SOA) has led to a number of proposals for service-oriented database architectures (SODA), e.g. [SZDG08, TB06]. The general idea of these architectures is very close to that of the above-mentioned CDBMS. The service-oriented DBMS (SDBMS) architecture [SZD07, SZDG08] has been defined based on the layered architecture proposed by Härder [Här05]. 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.

The CoBRA DB project [IDMW08, IFMW08] 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. Irmert *et al.* [ILN+09] present how a Transaction Manager can be added and removed at run-time using the d-APO approach, which has several disadvantages in terms of performance, code maintenance, limited functionality and testing [IFMW08].

In mobile applications, computing resources such as memory, processing capabilities, networking and power may all be limited and it is therefore important that a DBMS can be configured and optimized for a particular application setting. COMET [NNNH04] is a component-based real-time database for automotive systems that represents DBMS that can be statically configured for a given application or target device. FAME-DBMS [RSS+08] 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.

More recently, OctopusDB [DJ11] was proposed to restore the idea of a "one size fits it all" database system. OctopusDB is designed as a single system capable of mimicking different types of data management systems, such as OLTP databases, OLAP data warehouses, data stream management systems and search engines. In OctopusDB, the flexibility required to adapt to these scenarios is achieved by abandoning the assumption that a data management system is built around a central store. Instead of using a fixed store, OctopusDB collects all data in a central log that records insert and update operations as logical log entries. Based on the workload, OctopusDB transparently creates so-called storage views that represent all or part of the central log in a specific physical layout. As a single abstraction, storage views unify query optimisation, view maintenance, index selection and store selection. However, their use as a basis to extend, rather than adapt, databases has not yet been investigated.

In summary, although the need for flexible, customisable and configurable database systems has been recognised, none of the existing approaches focuses on supporting database research. Instead, their focus is entirely on adapting existing functionality rather than extending databases with new functionality. In this paper, we argue that

support for database research requires even more flexibility as it needs to be possible to introduce new concepts and functionality. Further, since many proposals adopt approaches in which specific components are replaced or specialised, it is difficult to support adaptation across many components 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 or components.

We conclude this section by noting that part of our work relates to the challenge of supporting domain-specific languages [MHS05, Fow10]. Apart from other contributions, our database research platform provides an environment that can be used to define domain-specific languages. This aspect of our work is reminiscent of Scala [OAC+04, OSV11] and its support for defining internal domain-specific languages. Although this feature has been used to support language-integrated queries [SZ09] in Scala, adapting and extending databases as well as providing a platform for database research are outside the scope of their work. Recently, domain-specific languages such as Pig Latin [ORS+08] and Hive [TSJ+09] have been proposed to facilitate processing of big data sets on large clusters using MapReduce [DG04]. Since the approach presented in this article addresses the challenges of supporting object database research, these approaches are out of the scope of this work. Finally, we point out that the area of adaptive middleware [SM03] and the ability to update components at run-time has been a long-standing research issue in the area of component systems and software engineering, referred to as business process adaptation, e.g. [PBJ98, MSKC04, BNS+05, SSH+05]. However, the focus there is mainly on issues of updating software components at run-time in terms of replacing executable code rather than on information management issues and, to the best of our knowledge, none of these approaches deal with altering or extending the metamodel itself, i.e. the object model in the case of object-oriented software components.

3 Challenges and requirements

We introduce three main challenges associated with building a platform for object database research. These challenges are motivated by possible database research goals and are therefore different from the requirements of production systems. On the one hand, such a platform must provide a high degree of flexibility in conceptual system design and support run-time changes to the data model. On the other hand, the need to manage large amounts of data or optimal run-time performance are less important at this stage. Since projects often combine several of these goals, a research platform needs to address all of them in order to be generally applicable. Furthermore, we argue that these challenges imply requirements that surpass what has been addressed by the approaches presented in the previous section.

3.1 Challenge 1: Development of novel database behaviour

Novel and emerging database application domains often present previously unaddressed requirements. For example, the advent of CAD/CASE systems [CW98] to manage complex design objects required novel database behaviour since traditional database transactions consisting of a small number of read and write operations were a poor

match to intricate design processes requiring large and long-running transactions. To address this, DBMS were extended with design transactions using a check-out/check-in paradigm rather than begin, commit and abort operations.

In the context of object databases, providing new database behaviour is a common goal of many research projects. For example, new requirements in database behaviour arise when interfacing event-based object-oriented programming with an object database. Traditionally, programming languages and databases employ two different and even conflicting event mechanisms. In the case of programming languages, events are defined, fired and handled. In contrast, databases provide the notion of triggers that are invoked by the system when predefined events occurs. In Section 6.1, we present a unified event model for consistent event-based systems.

3.2 Challenge 2: Support for new forms of data management

The second challenge of database research that we have identified is to provide new forms of data management. In the case of CAD/CASE systems, the management of large design objects and revision histories with multiple parallel branches were required. In order to address these requirements, DBMS were augmented with the notion of complex objects and versioning mechanisms.

For example, a new requirement arises from the fact that software is typically developed in a modular and incremental way. In contrast, the database application design process is sequential and often leads to monolithic systems, making it harder to respond to changing or evolving requirements. In Section 6.2, we present a novel data management construct that supports modular and step-wise database design.

3.3 Challenge 3: Exploration of different database architectures

New application domains often require databases to support novel forms of distribution and different network architectures. For example, information sharing in mobile settings has led to a variety of distribution architectures such as centralised, e.g. [EPS+01], decentralised, e.g. [XOW04], and semi-centralised client-server, e.g. [BBM+07].

In a layered architecture, distribution is implemented on top of the database supporting a particular architectural variant and mode of information sharing. A research platform should support investigations of how fundamental concepts for distribution and sharing could be integrated within a database. In Section 6.3, we show how we used our research platform to build a database that supports the development of applications with differing requirements for network architectures and information sharing.

3.4 Additional requirements

Addressing all three challenges leads to additional requirements not yet considered by previous work on configurable and tailor-made DBMS. As presented previously, these systems approach the problem of providing new functionality by adaptation of existing functionality. In particular, their approach typically entails the replacement of a general database component with a specialised component that implements the adapted functionality. This approach is valid if the results of a research project are mature, stable and general enough to benefit most applications. However, the nature of the challenges outlined above will often lead to new functionality that cannot be realised by adapting or replacing existing functionality. As argued in Steiner [Ste98],

supporting novel DBMS functionality is best done by introducing new concepts that are orthogonal to the existing data model. Orthogonal in this context means that these concepts are not restricted to specific constructs of the data model, for example only to tuples in the case of relational databases, but rather that the database developer can define the level of granularity on which to apply these concepts. Even if new functionality can eventually be implemented as an adaptation, prototyping this functionality as *extensions rather than adaptations* can be advantageous. By definition, research is experimental and exploratory in nature and, therefore, prototyping needs to be rapid and lightweight. In our experience, these requirements are more closely met by extensions, since they reduce upfront overhead and have better support for trial and error.

Another new requirement with respect to existing approaches is the need to support both *general and domain-specific extensions*. Both approaches have their advantages and disadvantages. Advances made through general database research potentially benefit a large number of applications, might not be able to address the specific requirements of a given domain. Vice versa, the setting of a particular application domain might also provide opportunities that can be leveraged to address its requirements more elegantly. The ability to do so is one of the main advantages of domain-specific database research. This requirement is especially pertinent in the context of the third challenge, where the exploration of different architectural styles precludes the approach of a standardised system architecture that is common to existing solutions. General solutions might use a client-server or standalone architecture that can be applied in a variety of use cases. In contrast, domain-specific application scenarios such as mobile and personal databases might require highly specialised architectures to deal with limited devices or special types of data transfer.

4 Metamodel extension modules

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 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 databases, 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) to support data definition, data manipulation and data retrieval.

Our approach assumes that data and metadata are handled uniformly, so that both the data model and the application model are represented explicitly as data. This means that, not only may all database functionalities such as storage management, query processing and constraint management be applied to metadata as well as data, but also that 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 adapted or extended.

We claim that many requirements imposed by database applications can be met by extending the metamodel with additional concepts. To support this, we employ a modular system metamodel, an overview of which is shown in Figure 1. The core database module $Module_{core}$ is shown on the left-hand side of Figure 1. It comprises

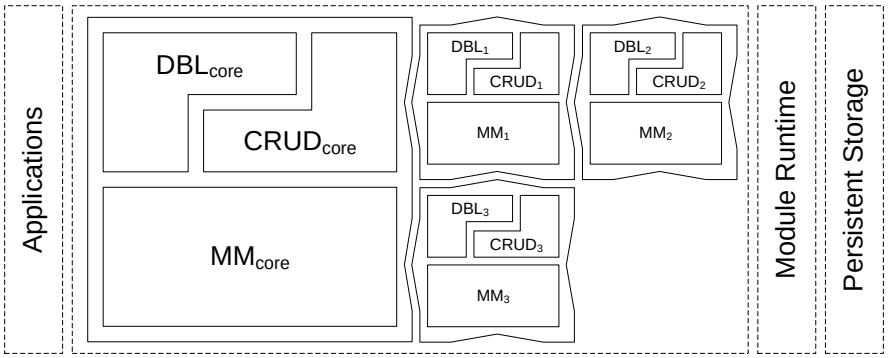

Figure 1 – Metamodel extension modules

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 concepts $\{C^1_{core}, \ldots, C^n_{core}\}$, such as *collection*, *object* and *attribute* in the case of an object database.

As seen in Figure 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 address 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 $\{C^1_{ext}, \ldots, C^m_{ext}\}$ where each concept $C^i_{ext} \in MM_{ext}$ extends or instantiates a concept $C^j_{base} \in MM_{core} \cup MM_1 \cup \ldots \cup MM_k$, formally written as $C^i_{ext} \lhd C^j_{base}$. 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.

The term $CRUD_{ext}$ refers to the operations that enable applications and developers 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 concept C_i to be created, retrieved, updated and deleted. When a module is loaded, its operator set $CRUD_{ext}$ is registered with the database so that they can be retrieved and used by an application or developer.

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 non-terminal 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 of our system also includes 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 since all other modules implicitly 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.

5 Platform to support research

After presenting the core metamodel based on the concepts introduced in the previous section, we describe the implementation of our platform and then show how it can be used by researchers.

5.1 Core metamodel

A core database module consists of a core metamodel MM_{core}, core management functionality $CRUD_{core}$ and a core database language DBL_{core}. Our approach works independently of a given data model 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 [NGD+08] and will present the details of the approach using this system as an example.

The concepts of the core metamodel MM_{core} are shown in Figure 2. It is based on the OM data model [Nor93] which integrates features of entity-relationship (ER) and object-oriented models. In comparison to more common modelling languages such as UML, OM has specifically been designed to model and manage object data and therefore provides concepts relevant to object database applications at its core rather than as extensions or profiles. OM uses a two-level model to clearly distinguish the typing and classification of entities. 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 in order to allow for objects with 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

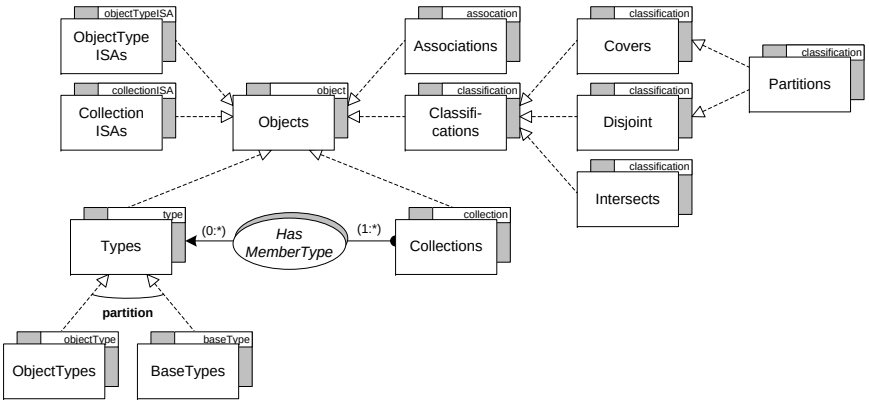

Figure 2 – Graphical representation of the core metamodel

classification constraints such as *disjoint*, *cover*, *partition* and *intersect* may be placed over these.

Graphically represented as shaded ovals, associations are a first-order concept of the model and defined as collections of pairs of values from the associated source and target collections. As in some extended ER models, cardinality constraints are specified by a minimum and maximum value that express the number of target objects to which a source object can be linked and vice versa. Associations can also be specialised over collections.

As can be seen in Figure 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.

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

While it is beyond the scope of this paper to present all aspects of the OM model in detail, we note that, based on the described metamodel, two different types of extensions can be distinguished as illustrated in Figure 3. On the left, we sketch a module that extends the core metamodel *by specialisation*. With this type of extension, tailor-made concepts for particular types of database systems can be realised by creating subtypes (and subcollections) of the core types (and collections) that specialise and, possibly, override the definitions of core concepts. On the right, we show a module that extends the core metamodel *by association*. In this case, genuinely new concepts are introduced and linked to existing concepts.

Figure 4 shows a UML class diagram of the manipulation operators for these concepts, which define create, retrieve, update and delete methods for instances of the corresponding type. For example, the creation of a collection takes the name and membertype of the collection as arguments, internally creates an object, dresses it with the collection type, sets the name and membertype attributes and returns the object. Given a collection object, its name and membertype can be retrieved with methods

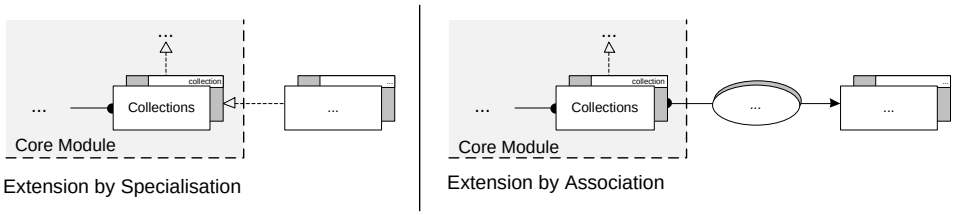

Figure 3 – Different types of metamodel extensions

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)

Figure 4 – UML class definitions of the core system operators

`getName()` and `getMemberType()`. 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 removes a collection object.

The third part of the core module is the database language DBL_{core}. Associated with the OM data model, the OML language [Lom06] 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.

```
1  /* data definition language */
2  create type contact ( name : string, phone : string );
3  create collection Contacts as set of contact;
4  /* data manipulation language */
5  $obj := create object;
6  dress $obj with contact ( name = "Fred Bloggs", phone = "(222) 555-4433" );
7  insert [ $obj ] into Contacts;
8  /* query language */
9  $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 (line 2), which is used as membertype for collection Contacts (line 3). 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 (line 5) and instantiated with the contact type using the `dress` operation (line 6). Then the object is inserted into the Contacts collection (line 7). Finally, a simple selection query over the Contacts collection is shown that selects the previously created object (line 9). 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.

```
statements      → statement { ";" statement }
statement       → [ ddl_statement | dml_statement | query_expression ]
ddl_statement   → create_statement
create_statement → "create" [ create_object | create_objecttype | ... ]
create_object   → "object"
create_objecttype → "type" name "(" attribute_list ")"
                  ...
```

Correspondingly, the DBL_{core} component is given by

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

5.2 Implementation

Our research platform is implemented in Java and uses Berkeley DB Java Edition[3] as a storage backend. On the right of Figure 1, the low-level persistent storage is shown. It is designed to provide persistent data management tailored to the requirements of the modular extension mechanism. The storage layer achieves its flexibility by means of the data model outlined in Figure 5. 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 value 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.

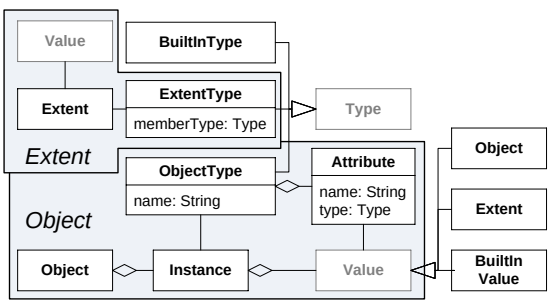

Figure 5 – Data model of persistent storage

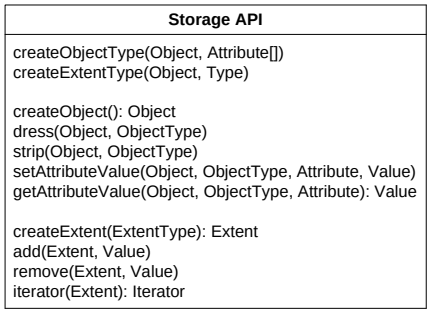

Figure 6 – Persistent storage API

The API of the storage component is shown in Figure 6. 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 through an iterator. Methods to delete types, objects and extents as well as query functionality are also provided, but not shown in the figure.

The entry point to our platform is the `DatabaseManager` and its interface is shown in Figure 7 along with the `Database` interface. The database manager manages all

[3] `http://www.oracle.com/technology/products/berkeley-db/`

Figure 7 – DatabaseManager and Database interface

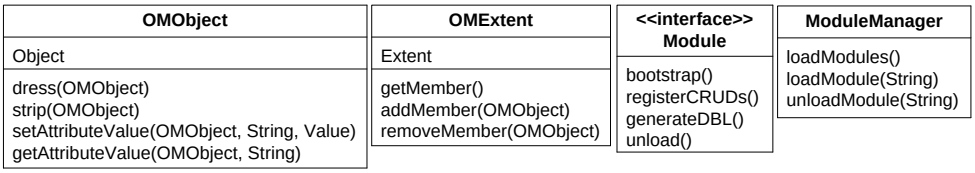

Figure 8 – Module Runtime

database instances and offers the functionality to create, retrieve, open, close and delete a database. Apart from other functionality, the `Database` interface provides methods to register and retrieve CRUD operators of metamodel extension modules. To manage the modules for each database, the database manager uses the `ModuleManager`, which is part of the module runtime shown in Figure 8. To orchestrate the lifecycle of modules, the module manager requires that developers of metamodel extension modules implement the `Module` interface, which is also defined by the runtime.

When a database is opened, the database manager invokes the module manager's `loadModules()` method, which first loads the core module and then loads all modules registered to be loaded automatically during initialisation of the database. For each module, the `loadModules()` method invokes the first three methods of the `Module` interface, namely `bootstrap()`, `registerCRUDs()` and `generateDBL()`. The `bootstrap()` method initialises the new metamodel concepts using the data definition operators provided by the core module. The `registerCRUDs()` method registers the CRUD operators and the `generateDBL()` method generates the database language extension. As a result of the loading process, the general metamodel is extended with the module's concepts and all of its database operations and language extensions are registered with the database. At run-time, the module manager allows single modules to be loaded and unloaded dynamically using the `loadModule()` and `unloadModule()` methods. The `unloadModule()` of the `ModuleManager` simply delegates its invocation to the `unload()` command of the corresponding module.

Apart from the `Module` and `ModuleManager` classes, the module runtime also provides the classes `OMObject` and `OMExtent`, which serve to wrap objects and extents managed by the persistent storage. Instances of `OMObject` uniformly represent application data objects and types as well as system metadata objects and types. It offers means to add and remove instances (`dress()` and `strip()`) as well as to read and write attribute values. `OMExtent` 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 run-time. Therefore, altering and extending the metamodel in the database does not require any changes to the in-memory Java representations of metamodel concepts.

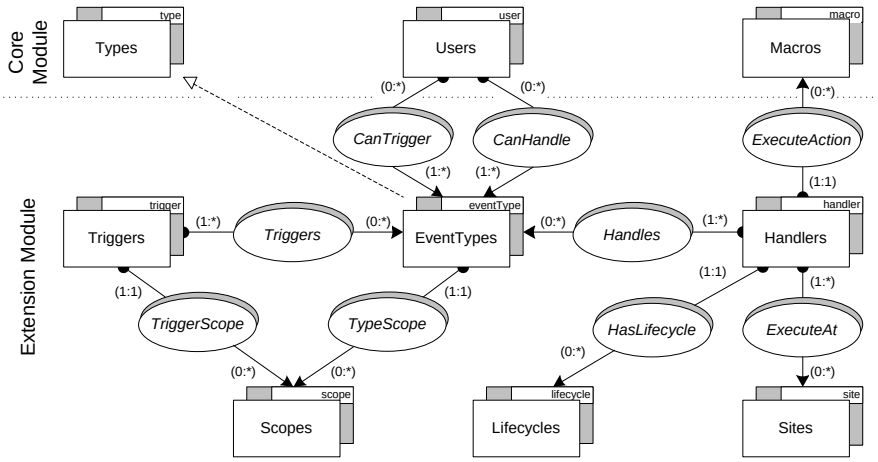

Figure 9 – Graphical representation of the event system metamodel

6 Case studies

To demonstrate the generality of our approach, we now present three case studies corresponding to the three types of research challenges presented in Section 3. The selection of case studies also reflects application scenarios in which object databases are typically used, such as embedded and mobile settings. Therefore, object databases are often used as single-user databases with an emphasis on tight integration with an object-oriented programming language. The first and the second case study both highlight how our platform was used to improve this integration. While the first case study demonstrates the unification of event-based programming and database triggers, the second focuses on supporting techniques of modular design and reuse known from object-oriented programming directly within object databases. Finally, the third case study is situated in the area of mobile information systems and uses our platform for object database research to design and validate concepts that facilitate the development of such applications. We report on the first case study in detail to document all aspects of defining and creating a metamodel extension module, while the other two focus on specific points of interest.

6.1 Event-based programming

Event-based programming is a popular paradigm for decoupling components and enabling asynchronous processing. The underlying concept of automatically invoking actions in response to pre-defined events is well-known in database systems as triggers. Since object databases are tightly coupled with programming languages, it is desirable to also support event-based programming directly in the database. In previous work [GNS07, GLdN09], we investigated how concepts from programming languages and database triggers could be unified into a general and flexible event model for object databases. Our model is capable of supporting a rich variety of application requirements common in emerging domains such as sensor databases as well as more traditional applications. Further, it also supports distributed triggers where an event in one database may trigger an action in another database.

Using our platform, we have implemented the event model as a metamodel extension module. The corresponding metamodel shown in Figure 9 defines how the three core concepts—EventTypes, Triggers, and Handlers—interact to provide the desired functionality. The type eventType is a specialisation of the type type defined by the core module and defines the list of parameters that are passed to the handler when events of this type are triggered. Instances of type eventType are linked to user instances through associations canTrigger and canHandle to control who can trigger and handle events of this type. Given sufficient user permissions, applications can then trigger and handle events of a certain type using the event module's CRUD classes or its DBL extension, which will both be discussed later in this section. Finally, eventType instances are associated with a scope instance, which defines the set of objects targeted by the event. Type handler defines a condition which guards the execution of the associated macro, i.e. OM's notion of a stored procedure. Since our event system is designed for distributed settings, a handler can be associated with any number of site instances that define where the event action will be executed. Finally, handlers are also associated with a lifecycle instance, which can either be a time span, a point in time or a number of notifications. When a handler is notified, the condition is evaluated and, if it returns true, the macro is executed. Instances of type trigger record the operation that has occurred, the object on which the operation has happened, a list of parameter values passed to the handler and the user that executed the operation. Furthermore, a multiplicity attribute specifies whether this trigger is fired once or multiple times, and, in the latter case, with which period and for how many times or for how long.

Based on the event system's metamodel, we define the three components of the metamodel extension module as follows. The first part, the module's metamodel, is simply the sum of its types, collections and associations.

$$\begin{aligned} MM_{event} \quad = \quad & \{\{\mathit{eventType}, \mathit{trigger}, \mathit{handler}, \ldots\}, \\ & \{\mathit{EventTypes}, \mathit{Triggers}, \mathit{Handlers}, \ldots\}, \\ & \{\mathit{Triggers}, \mathit{Handles}, \ldots\}\} \end{aligned}$$

Note that all collections, if not indicated otherwise in Figure 9, are subcollections of the Objects collection (not shown in the figure) of the core metamodel. The operators providing the creation, retrieval, update and deletion of these metamodel concepts are defined as

$$CRUD_{event} \quad = \quad \{\mathit{EventTypes}, \mathit{Triggers}, \mathit{Handlers}, \ldots\}.$$

A subset of these operations, which are provided to manage and configure our event system, are illustrated in Figure 10. Note that the implementations of the event system operators rely on the presence of the core system operators in the same way the event metamodel extends the core metamodel. For example, the creation routine of `EventTypes` will invoke the create method in `Types` to create a general type and then dresses it with the eventType type.

To interact with our event system at the level of a programming language, we have defined DBL_{event}, which is an extension of DBL_{core}, i.e. OML as presented in the previous section, to provide constructs to trigger and handle events. The simple example below illustrates the definition of an event type, trigger and handler. The trigger is fired whenever an instance of type person changes. The event type defines the person's name as the payload and the scope to be type person. It also grants trigger access to local users and handler access to any user. The handler simply prints

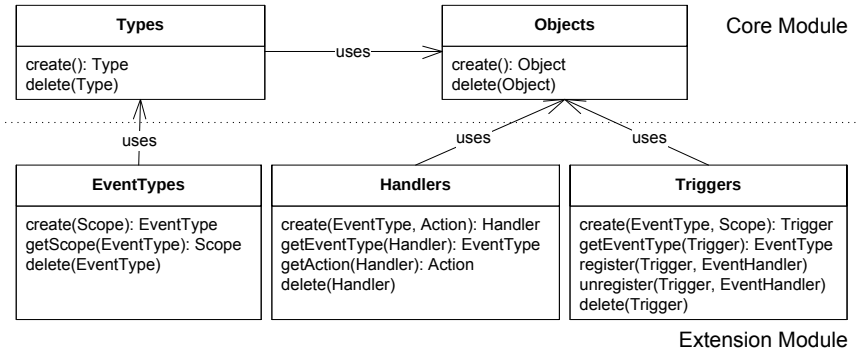

Figure 10 – UML class definitions of the event system operators

that name to the console. To do so, we use an inline command, rather than a macro. Finally, we add an `onchange` event to the person type that fires the trigger, passing along the name of the person that has changed.

```
1  /* Create event type */
2  create eventtype person_change ( name: string ) scope type person
3     enable trigger local.*, handler any.*;
4  /* Create handler */
5  create handler person_change_handler for person_change
6     condition ( ... )
7     exec ( print "Person " + name + " changed." )
8     lifecycle once;
9  /* Create trigger and link to person type */
10 alter type person (
11    onchange create trigger for person_change fire ( $self.name );
12 )
```

Formally, the language sketched above is defined as the set of production rules P_{event} below. We limit ourselves to a subset to illustrate the approach.

create_statement → "create" [... | create_eventtype | create_handler | create_trigger]
create_eventtype → "eventtype" *name* "(" attribute_list ")" scope_def [access_def]
create_handler → "handler" *name* "for" eventtype_ref [condition_def]
"exec" (macro_ref | script) [lifecycle_def]
create_trigger → "trigger" [*name*] "for" eventtype_ref
"fire" "(" value_list ")" [multiplicity_def]
...

Together with the terminals and non-terminals, the productions above are the basis for the definition of the database language component DBL_{event} as

$$DBL_{event} = \{\{\text{create\_eventtype}, \text{create\_handler}, \text{create\_trigger}, \ldots\}, \{\texttt{"eventtype"}, \texttt{"handler"}, \texttt{"trigger"}, \texttt{"scope"}, \ldots\}, P_{event}\}.$$

Once the metamodel extension, the CRUD operators, and the database language have been defined, the module needs to be realised using the interfaces of our platform to support object database research. In order to give a sense of the development effort

involved and to show how a developer would use the platform, we briefly outline the implementation of the event module in terms of source code examples. We begin by presenting the module's main class, which defines the module's metamodel, and then outline the create method from the CRUD operator for type eventType.

The main class `EventModule`, which implements the `Module` interface shown in Figure 8 in Section 5, is summarised in the code listing below. The first method, `bootstrap()`, creates the metamodel of the module. In the listing, we outline the code that creates the metadata concepts for type eventType. The statements on lines 5 and 6 create definitions for attributes `name` and `scope`, respectively. Attribute `name` is of type `STRING` and `scope` is of type `OID`. Since both of these types are defined by the core metamodel module, they can be retrieved using the CRUD operator for objects of type namedObject, which is accessible through method `namedObjects()`. Once these attribute definitions have been created, the type eventType itself is created on line 7. The main class of a metamodel extension module also registers the CRUD operators with the database. This task is performed by method `registerCRUDs()` (line 11) and the call to register the CRUD operator for type eventType is shown on line 13.

```
1  public class EventModule extends AbstractModule implements Module {
2
3     public void bootstrap(OMTransaction tx) {
4        /* ... */
5        OMStructuredValue nameAttr = this.getDatabase().attributes().create(tx,
             "name", this.getDatabase().namedObjects().retrieve(tx,
             Schema.STRING));
6        OMStructuredValue scopeAttr =
             this.getDatabase().attributes().create(tx, "scope",
             this.getDatabase().namedObjects().retrieve(tx, Schema.OID));
7        this.getDatabase().objectTypes().create(tx, "eventType", new
             OMStructuredValue[] { nameAttr, handlersAttr });
8        /* ... */
9     }
10
11    public void registerCRUDs() {
12       /* ... */
13       this.getDatabase().registerCRUD(new EventTypes(this.getDatabase()));
14       /* ... */
15    }
16
17    /* ... */
18
19 }
```

Users and developers can interact with the concepts of a metamodel extension module either imperatively using the CRUD operators or declaratively using the DBL. The listing below contains a code excerpt of the CRUD operator for type eventType. We show the `create()` method that instantiates a new event type object and initialises its fields. On line 4, we begin by creating a new object[4]. To instantiate the new object with type eventType, we use method `dress()` on line 6. Once the object has been dressed with a certain type, the corresponding attribute values can be set on lines 7 and 8. Finally, we return the newly created event type on line 9. The `browse()`

[4]Since our database system uses multiple inheritance, creating a new object consists merely of creating a new object id.

method returns a façade that corresponds to the interface `EventType`, which provides easy and non-generic access to the event type's fields and methods.

```
1  public class EventTypes {
2
3     public EventType create(OMTransaction tx, String name, Scope scope) {
4        OMObject eventType = this.getDatabase().objects().create(tx);
5        OMObject eventTypeType = this.getDatabase().namedObjects().retrieve(tx,
             "eventType");
6        eventType.dress(tx, eventTypeType);
7        eventType.setAttributeValue(tx, eventTypeType, "name", name);
8        eventType.setAttributeValue(tx, eventTypeType, "scope",
             scope.getBaseObject());
9        return eventType.browse(EventType.class);
10    }
11
12    /* ... */
13
14 }
```

In order to register the main class of a metamodel extension module, a so-called module description file needs to be provided. The XML syntax of module description files is illustrated in the following example. Note that every module is identified by a unique name and can therefore be cross-referenced in module description files. The description file also contains information about module dependencies and other options. For example, the load policy controls when a module is initialised. Since the event module is a central module used by many other modules, it is loaded right when a database is opened. Finally, state management can be set to persistent or transient to control whether or not a module's metadata is retained after the module is unloaded.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <module xmlns="http://www.globis.ethz.ch/avon/modules" version="1.2">
3     <name>avon-event</name>
4     <main-class>EventModule.class</main-class>
5     <dependencies>
6        <dependency>avon-core</dependency>
7     </dependencies>
8     <load-policy>DB_OPEN</load-policy>
9     <state>persistent</state>
10 </module>
```

As can be seen from these examples, the application programming interface of our platform is very generic and conceptually situated at the level of the metamodel. In order to provide additional concepts and functionality, a developer needs to "meta-program", i.e. instead of directly instantiating classes, generic interface methods are used. While this approach leads to more verbose code that is slightly more challenging to maintain, it also has several important advantages that justify our choice. First, this generic approach provides the degree of flexibility required in a platform to support research. As data model concepts are not cast into compiled classes, the model can be dynamically evolved at run-time by changing the system metamodel. Second, the resolution of module dependencies can be postponed from compile-time to run-time. An extension module that extends another module does not need to access the source or compiled code of the base module. It only requires that the base module is available at run-time and that its metamodel concepts are known by name. In our approach, it

is therefore the metamodel that defines the contract between the various modules of the database system. This characteristic stands in strong contrast to other approaches that rely on interface definitions to encapsulate system components.

The event module was developed by a graduate student in the context of a master thesis of six months duration. The rich event model provided by the module has proven to be an enabling technology for several other database extensions, for example, the mobile information sharing module presented in Section 6.3. As a result of its success, the event module has been integrated with and is now maintained as part of the main distribution of OMS Avon.

6.2 Information components

While modular and incremental development is a state-of-the-art practice in software engineering, information systems are still developed sequentially, resulting in a monolithic system that is hard to modify and adapt. In Leone *et al.* [LGN11], we introduced the concept of *information components* as a mechanism to support the creation and composition of an information system in a modular manner based on a sharing and reuse paradigm. The unit of reuse is an information component that comprises both metadata and data. If only metadata is present, then the composition is at the schema level to support the design process. Optionally, data may be included to support data reuse. Sharing and reuse is realised by means of queries over well-defined component export interfaces, which, for a specific component, specify the data and metadata offered for reuse. By providing means for fine-grained reuse of metadata and data across information components, new information components can be created by making use of existing ones and combining them in new ways.

Figure 11 gives an example of a composition scenario. Assume that a small retail company manages products and customers. On the left hand side, we illustrate two components, one for customer management and one for product management. Assume the company now decides to sell their products online. They could reuse the metadata and data of the customer and product components, and compose a new online store component as illustrated on the right hand side. Queries over component export interfaces define the reuse of both metadata and data from existing components. To support reuse of existing components, export interfaces are registered with a component registry, which can be browsed by developers. The definition and registration of a component's export interface is part of the component definition process.

Figure 12 gives an overview of the metamodel extension for information components. An information component defines information elements for metadata and data. Metadata objects are instances of the metamodel concepts of the core module, modelled through the subcollection relation between Metadata and Objects, and, consequently,

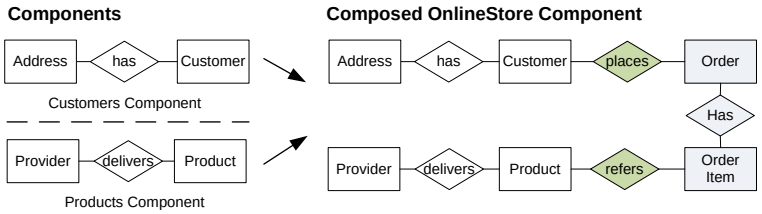

Figure 11 – Composition Scenario

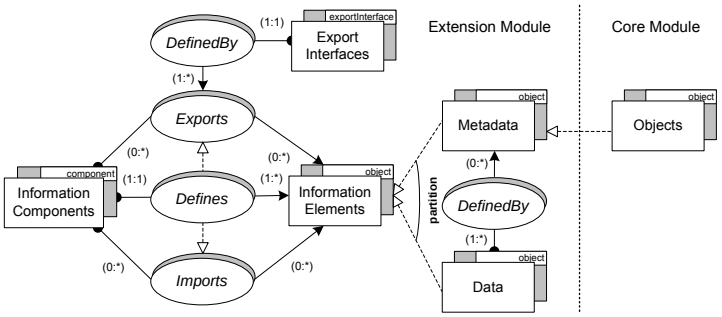

Figure 12 – Graphical representation of the information components metamodel

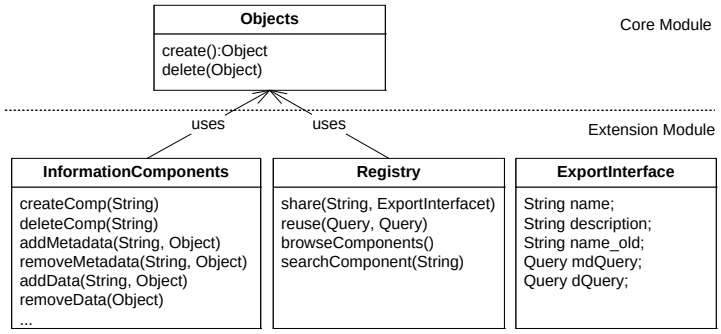

Figure 13 – UML class definition of information components operators

information components extend the core by specialisation. The metadata objects define the structure of the data objects, as indicated by the DefinedBy association between Metadata and the Data.

The reuse of information elements is reflected by the Import association, which is a subassociation of the Defines association between the InformationComponents collection and the InformationElements. The Import association defines, for a given information component, the information elements that have been imported from other information components. A component may also offer metadata and data for sharing. This is reflected by the Export association between InformationComponents and InformationElements, which is also a subassociation of Defines. Information element export is specified by a query that declares the information elements to be reused. Note that an information component can define multiple such export queries.

Figure 13 gives an overview over the operators that can be used to create and manipulate instances of metamodel constructs. The InformationComponents operator supports the creation, retrieval, deletion, composition and manipulation of information components. Sharing and reuse functionality is provided by the Registry operator. Components can be made available for reuse using the method `share()` that takes two input parameters—the component name (which may be a new name due to renaming) as well as an export interface. ExportInterface objects encapsulate all information needed to support component export, such as a possible renaming, a component description used when browsing the registry and inspecting the component as well as queries over the metadata and data to be exported. Information components may import metadata and data from the registry using the `reuse()` method that again

takes two input parameters, the queries over the metadata and data. Component reuse is realised by reference and is simply a view on local metadata and data objects and thus supported by OMS Avon.

The first version of the information component module was developed by a graduate student as part of a six month master thesis. It was used to build a platform supporting the collaborative design and development of personal information management systems [LGN11]. Non-expert users can select provided components and compose them graphically to construct their personal information space. A refined and generalised version of the module was used as a basis to realise a community-based design and development platform for eCommerce systems [LN11]. Developers can build tailored eCommerce solutions based on a set of components that represent basic eCommerce building blocks, and can be selectively customised and composed.

6.3 Mobile information sharing

Existing development platforms for mobile applications lack integrated support for information sharing, which means that developers need to address these requirements repeatedly and at a low-level. As an example, consider a mobile application that allows tourists to enter and share reviews for places that they visit. One way of realising this functionality is to have a central server, while another would be to share the reviews via kiosk servers at points of interest, using Bluetooth or WiFi ad-hoc connectivity to automatically transfer reviews about a location to and from the tourists' mobile devices. Existing approaches tend to build a layer that implements a specific distribution model *on top* of traditional database systems. In contrast, our aim was to *integrate* concepts into a database that provide native support for data sharing, while allowing developers the flexibility of configuring their own distribution models and collaboration logic.

Our approach introduces a general notion of *shared collections*, where the collaboration logic can easily be configured. To validate the approach, we implemented it in OMS Avon as a metamodel extension module. Details of the shared collection concept and how it supports different modes of data sharing are given in [dGN09, de 10]. Here, we will focus on showing how shared collections could be embedded in an object-oriented development environment by leveraging the language extension of the metamodel extension module.

Shared collections extend data collections and introduce functionality to share members with collections of the same name in other databases that can be located on different devices. We use the form *Name* to refer to a collection by its name and $Name\langle T\rangle$ if the member type is of importance. On all devices, the shared collection $C\langle T^{local}\rangle$ is configured so that it is connected to collections with the same name C and compatible member type T^{remote} residing on other devices in physical proximity. Member types are compatible if they are the same or if T^{local} is a supertype of T^{remote}.

Shared collections provide *share* operations for the sharing of a single, multiple or all members, each with a single or multiple peers. Also, *setAvailable* operations are used to turn on the availability to receive members, which can be unconstrained or restricted to a single or set of peers. Both *share* and *setAvailable* methods are used to configure the sharing behaviour in terms of when and which members are shared with which peers. Consequently, different distribution architectures and modes of data sharing can be realised by configuration rather than implementation.

Table 1 summarises the collaboration configurations and operations, their arguments and example values. *Data filters* use queries to specify, which members should

Table 1 – Shared Collection Configurations and Operations

Configuration	**Arguments**	**Examples**
Data Filter	Query	Members created locally
		Members related to recipient
Trigger	Event, Condition, Action	User action
		Data creation or manipulation
		Connection state
Sharing Mode	Transmission Semantics	Copy, Reference
	Durability	Persistent, Transient
Neighbourhood	Peers	Peer B on A and C
		Peers A and C on B
Operation	**Arguments**	
setAvailable	Configuration	
share	Member(s), Peer(s), Configuration	

be shared. A *trigger* specifies whether a sharing processes is initiated by user actions, collection updates or peers appearing in physical proximity. The *sharing mode* specifies whether shared members should be transferred with copy or reference semantics, and if data should persist after disconnection. Finally, each shared collection is associated with a set of peers that form its *neighbourhood*. Members may be shared with one particular peer or be broadcast to all peers in the neighbourhood.

We leverage the fact that metamodel extension modules can define a language extension to introduce shared collections as a first-order concept in our database language. While this implementation has served as a proof of concept, we note that it would also be possible to extend existing languages such as Java, Scala, or C++ with the shared collection concept, either by using a pre-compilation step or language extensions. Using what we call "collection-oriented programming", developers implement a mobile information system by specifying domain classes, collections of persistent and/or shared data and configuring the persistent and sharing services to meet their requirements. Below we show example source code for the case of the tourist application. As shown on line 1, classes are created for domain entities such as reviews in the usual way. Line 3 then defines the collection of reviews, where the keyword `persistent` declares that the collection members are to be stored persistently and the three dots denote the definitions of the remaining characteristics such as methods, queries and event handling. Data filters specifying a set of members to be shared are defined using the `query` keyword to attach a query to a collection. For example, we could attach two queries to the collection of reviews, one to access reviews authored by the user (line 4) and the other to access reviews about the current location (line 5). Similarly, an event handler is attached to the collection on line 6, where the type of event handled is specified and the three dots indicate the handler implementation.

```
1  class Review { String text; User author; ... }
2
3  persistent collection Reviews<Review> {
4     query AuthoredByUser() { ... }
5     query AboutLocation() { ... }
6     handle AddEvent(Event event) { ... }
7  }
```

Based on this skeleton, it is possible to configure our tourist application with the following sharing behaviour. When a tourist comes close to a kiosk server, a new peer object is added to the system collection called `Neighbourhood` on both the mobile device and the server. On the server, the addition is handled by executing the `AboutLocation` query and sharing the result of this query with the mobile device. Conversely, on the mobile device, the addition is handled by executing the `AuthoredByUser` query and sharing the result with the server. In this way, the tourist receives reviews from other users, while their own review is exchanged with and stored on the server. The server collection of reviews would be configured to make them persistent, whereas the collection on the mobile device could be configured to have reviews received transient so that the data is deleted once they leave the location and are no longer connected to the server.

Using OMS Avon together with the shared collection module, the tourist application was implemented with less effort than would be required using state of the art mobile application development platforms including the Apple iPhone, Google Android and Microsoft Windows Phone SDK. Since all of the recurring requirements such as information persistence and sharing as well as proximity detection are addressed once from within OMS Avon, they do not need to be addressed by the application developer for each application. Moreover, based on the general collaboration concepts introduced by the shared collection module, different distribution models and modes of information sharing can be realised by means of configuration rather than implementation. As a result, we were able to investigate a number of different application scenarios including a recommender system where spatio-temporal proximity was used as the basis for user similarity in collaborative filtering [dSNG08].

6.4 Further examples

To show the range of projects that we have been able to support using our research platform, we conclude this section by mentioning two other projects.

Context-awareness is an important topic in web engineering and, in earlier research, we investigated models and mechanisms to support this in content management systems [BDG+05, GN07]. To take this work further, we re-implemented the approach as a metamodel extension module in OMS Avon and developed a domain-specific language, XCML, for context-aware web applications [NGLN10, NGLN12].

Object databases traditionally represent application logic by means of methods tightly bound to objects through their type definitions. We wanted to investigate means of being able to define and reuse behaviour more generally and flexibly, for example across type definitions, or for that behaviour to evolve over time. In Leone *et al.* [LNSd09], we introduce a notion of role-based services that support the dynamic binding of active content to database objects. As proof of concept, the concept was implemented using a metamodel extension and applied to personal resource management to allow different services to be used in different contexts.

7 Discussion

In this section, we begin by validating our approach with respect to the challenges and requirements established in Section 3. We then discuss experiences gained from applying our platform in different projects, outline best practices derived from those experiences, and address limitations of our approach.

Table 2 – Classification of the case study examples

Case Study Module	Database Behaviour	Data Management	Different Architectures
Event-based programming	✓	–	–
Information components	–	✓	–
Mobile information sharing	✓	–	✓
Web content management	✓	✓	–
Role-based services	✓	–	–

7.1 Validation

To validate our approach, we return to the challenges of the object database research space introduced in Section 3. In Table 2, we classify the projects presented in the previous section with respect to the types of research challenges addressed.

The metamodel extension module for event-based programming is an example of introducing new behaviour into an object database. As a consequence, the focus of the module lies on the CRUD operators that implement this new behaviour and the language extension that exposes it to programmers. While the metamodel shown in Figure 9 introduces several new concepts, the goal of these concepts is to configure the behaviour of the module, rather than to introduce new forms of data management.

In contrast, the main focus of the metamodel extension module for information components is providing new forms of data management. The functionality provided by the CRUD operators and language extension supports the management of information components by defining new data definition and manipulation operations that allow information components to be managed as well as shared and reused.

Finally, the metamodel extension module for mobile information sharing shows how the platform can support experimentation with new distribution architectures and modes of information sharing. While the focus of this project was clearly on architectural challenges, the module also introduces a significant amount of new database functionality that governs information sharing in a mobile setting. However, a large portion of this additional database behaviour was actually provided by the module for event-based processing, which demonstrates how the module concept allows research itself to be modularised.

We also noted in Section 3.4 that a research platform for object database technologies needs to meet additional requirements. The first of these is that both adaptation and extension should be supported. We have described how adaptation is supported by specialisation of concepts in the core model, whereas extension is supported by association of new concepts. For example, in the case of mobile information sharing, the concept of a shared collection was introduced as a specialisation of the core collection concept to add functionality, while, in the case of the event module, the concept of an event handler was introduced by association to the macro concept of the core model.

The second requirement is that a research platform needs to support both domain-specific and general database research. Revisiting the case studies presented in the previous section, we can observe that our approach of using metamodel extension modules does not bias the proposed research platform in this respect. The modules for event-based programming, role-based services and information components are all examples of contributions resulting from general database research. In contrast, the module for web content management is domain-specific to the area of web engineering

in the same way as the module for mobile information sharing is specific to the area of mobile and pervasive computing.

7.2 Experiences and best practices

In the following, we report on experiences gained while using the platform to realise the implementation of numerous projects, ranging from short-term student works to PhD theses and even larger projects.

Apart from enabling and fostering our research, the presented approach also proved beneficial in terms of organising, structuring and managing the work involved in designing and developing new database technologies. At any point in time, our object database was used productively in both education and research, while being extended and adapted in parallel for new application domains. Even with meticulous release planning and advanced software revision systems, managing a project of this nature is very challenging. By building the metamodule extension module mechanism into the database system itself, we were able to isolate different projects from the core database system as well as from each other. Thereby, it was possible to support experimentation and prototyping without compromising the stability and development of the core database system. A similar separation could be achieved by creating code branches for every research project in a revision control system. We believe that our solution is more elegant because, on the one hand, modules intrinsically have the right granularity in comparison to branches and, on the other hand, they do not require complicated merges once a project is completed. In our platform, a stable and complete module is simply included in the main distribution and activated on demand.

The flexibility and generality of our platform implies that developers are faced with a number of possibilities as to how they can realise a project. For example, a major design decision is whether to integrate the new concepts into the existing database in a tight or loose manner. In our approach, tight integration corresponds to extension by specialisation, which refines or overrides existing concepts. In contrast, loose integration corresponds to extension by association, which simply links new concepts to existing ones. As shown by the case studies presented, it is also possible to mix these two approaches in a single metamodel extension module. In order to support users of our research platform, we have defined a number of best practices that are intended to guide this decision. As mentioned above, one factor that can influence this choice is whether the module adapts existing or introduces new functionality. The distinction between general and domain-specific research is another indication of how to design a module. In our experience, modules that implement the results of general research projects benefit from tight integration with the core system, whereas domain-specific modules can be more loosely attached. Finally, the development life-cycle of a module also impacts the decision. New and experimental modules that are still being prototyped are best realised as loose extensions, while stable modules that are ready to be deployed can be more tightly integrated. As a consequence, we have seen several modules that transitioned with respect to their integration approach over time.

7.3 Limitations of the approach

We conclude the discussion of our approach by characterising possible limitations and corresponding solutions. Extending and adapting a database system at the metamodel level rather than at the level of an application programming interface affects the

run-time performance. In contrast to a more strongly coupled approach, our generic approach that is motivated by the required flexibility implies that most calls translate to a number of metamodel queries. Therefore, the great flexibility of our approach can be a disadvantage in a production setting. Additionally, the direct applicability of our approach in commercial object databases is limited by the fact that these systems often do not expose a well-defined metamodel, which can be extended. However, the role of our platform is to support the design, development, and validation of new concepts and functionality. Once these new technologies have proven successful and useful, they are ported to a specific production system, where they can be cast into that system's data model and optimized for performance.

The implementation of the platform itself imposes certain limitations. While the storage model that we provide is very general, it ultimately is a row-oriented storage and will therefore render experimentation with column-oriented technologies difficult. One possibility to avoid this limitation would be to implement a column-oriented view on top of our row-oriented storage in analogy to Bruno [Bru09]. Another, more radical, solution could be to replace our current storage with a back-end similar to the one proposed by OctopusDB [DJ11].

At the time of writing this article, we have not yet fully addressed the challenges of loading and unloading modules in a general way. In contrast to pure software modules, metamodel extension modules can depend on a persistent state. In some cases, unloading a module implies that its state is removed, while in other cases the module's state cannot safely be removed. Currently, module developers can specify whether the state of a module is transient or persistent in the module description file. Additionally, they can control the management of the persistent state in terms of the `bootstrap()` and `unload()` methods in `Module`. In the future, we hope to provide more support for this issue by identifying modules that share common requirements and can therefore be handled in a uniform way.

8 Conclusion

We motivated the need for research platforms to address the requirements of novel database applications. Moreover, we argued that the support currently offered by so-called customisable or tailor-made databases is insufficient since they are not designed to support experimentation and the prototyping of new concepts and functionality.

Our approach to building a research platform evolved from our previous work on adaptive data management and is based on the notion of metamodel extension modules, rather than the configuration of components within a standardised architecture. Metamodel extension modules consist of a metamodel that defines new concepts, an API and implementation to manipulate and interact with the new concepts, and an optional database language.

We demonstrated the use of the platform and its generality by presenting a variety of research projects that were successfully conducted using our platform. These projects covered the three main dimensions of database research, namely development of novel behaviour, support for new forms of data management and exploration of different architectures. To facilitate the applicability of our work outside our research, we also discussed lessons learnt from our experiences in terms of best practices and limitations of the approach.

References

[ARSS08] Sven Apel, Marko Rosenmüller, Gunter Saake, and Olaf Spinczyk, editors. *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*. ACM, 2008. doi:10.1145/1385486.

[BBM+07] Arianna Bassoli, Johanna Brewer, Karen Martin, Paul Dourish, and Scott Mainwaring. Underground Aesthetics: Rethinking Urban Computing. *IEEE Pervasive Computing*, 6:39–45, 2007. doi:10.1109/MPRV.2007.68.

[BDG+05] Rudi Belotti, Corsin Decurtins, Michael Grossniklaus, Moira C. Norrie, and Alexios Palinginis. Interplay of Content and Context. *J. Web. Eng.*, 4(1):57–78, 2005.

[BNS+05] Jaiganesh Balasubramanian, Balachandran Natarajan, Douglas C. Schmidt, Aniruddha S. Gokhale, Jeff Parsons, and Gan Deng. Middleware Support for Dynamic Component Updating. In *Proc. Intl. Symp. on Distributed Objects and Applications (DOA)*, pages 978–996, 2005. doi:10.1007/11575801_4.

[Bru09] Nicolas Bruno. Teaching an Old Elephant New Tricks. In *Conf. on Innovative Data Systems Research (CIDR)*, 2009.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998. doi:10.1145/280277.280280.

[de 10] Alexandre de Spindler. *A Collection-Oriented Application Framework for Mobile Information Systems*. PhD thesis, ETH Zurich, Switzerland, 2010. doi:10.3929/ethz-a-6620201.

[DG01] Klaus R. Dittrich and Andreas Geppert, editors. *Component Database Systems*. Morgan Kaufmann, 2001.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. Symp. on Opearting Systems Design & Implementation (OSDI)*, pages 137–149, 2004.

[dGN09] Alexandre de Spindler, Michael Grossniklaus, and Moira C. Norrie. Development Framework for Mobile Social Applications. In *Proc. Intl. Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 275–289, 2009. doi:10.1007/978-3-642-02144-2_24.

[DJ11] Jens Dittrich and Alekh Jindal. Towards a One Size Fits All Database Architecture. In *Conf. on Innovative Data Systems Research (CIDR)*, pages 195–198, 2011.

[dSNG08] Alexandre de Spindler, Moira C. Norrie, and Michael Grossniklaus. Recommendation Based on Opportunistic Information Sharing Between Tourists. *J. of IT & Tourism*, 10(4):297–311, 2008. doi:10.3727/109830508788403178.

[EPS+01] Fredrik Espinoza, Per Persson, Anna Sandin, Hanna Nyström, Elenor Cacciatore, and Markus Bylund. GeoNotes: Social and Navigational Aspects of Location-Based Information Systems. In *Proc. Intl. Conf. on Ubiquitous Computing (Ubicomp)*, pages 2–17, 2001. doi:10.1007/3-540-45427-6_2.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[GLdN09] Michael Grossniklaus, Stefania Leone, Alexandre de Spindler, and Moira C. Norrie. Unified Event Model for Object Databases. In *Proc. Intl. Conf. on Object Databases (ICOODB)*, pages 113–131, 2009. doi:10.1007/978-3-642-14681-7_7.

[GLdN10] Michael Grossniklaus, Stefania Leone, Alexandre de Spindler, and Moira C. Norrie. Dynamic Metamodel Extension Modules to Support Adaptive Data Management. In *Proc. Intl. Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 363–377, 2010. doi:10.1007/978-3-642-13094-6_29.

[GN07] Michael Grossniklaus and Moira C. Norrie. An Object-Oriented Version Model for Context-Aware Data Management. In *Proc. Intl. Conf. on Web Information Systems Engineering (WISE)*, pages 398–409, 2007. doi:10.1007/978-3-540-76993-4_33.

[GNS07] Michael Grossniklaus, Moira C. Norrie, and Julian Sgier. Realising Proactive Behaviour in Mobile Data-Centric Applications. In *Proc. Intl. Workshop on Ubiquitous Mobile Information and Collaboration Systems (UMICS)*, pages 561–575, 2007.

[Här05] Theo Härder. DBMS Architecture – New Challenges Ahead. *Datenbank-Spektrum*, 14:38–48, 2005.

[IDMW08] Florian Irmert, Michael Daum, and Klaus Meyer-Wegener. A New Approach to Modular Database Systems. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 40–44, 2008. doi:10.1145/1385486.1385498.

[IFMW08] Florian Irmert, Thomas Fischer, and Klaus Meyer-Wegener. Runtime Adaptation in a Service-Oriented Component Model. In *Proc. Intl. Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS)*, pages 97–104, 2008. doi:10.1145/1370018.1370036.

[ILN+09] Florian Irmert, Frank Lauterwald, Christoph P. Neumann, Michael Daum, Richard Lenz, and Klaus Meyer-Wegener. Semantics of a Runtime Adaptable Transaction Manager. In *Proc. Intl. Database Engineering & Applications Symposium (IDEAS 2009)*, pages 88–96, 2009. doi:10.1145/1620432.1620442.

[LGN11] Stefania Leone, Matthias Geel, and Moira C. Norrie. Managing Personal Information through Information Components. In *Information Systems Evolution*, pages 1–14. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-17722-4_1.

[LN11] Stefania Leone and Moira C. Norrie. Building eCommerce Systems from Shared Micro-Schemas. In *Proc. Intl. Conf. on Cooperative Information Systems (CoopIS)*, pages 284–301, 2011. doi:10.1007/978-3-642-25109-2_19.

[LNSd09] Stefania Leone, Moira C. Norrie, Beat Signer, and Alexandre de Spindler. From Static Methods to Role-Driven Service Invocation – A Metamodel for Active Content in Object Databases. In *Proc. Intl. Conf. on Conceptual Modelling (ER)*, pages 444–457, 2009. doi:10.1007/978-3-642-04840-1_33.

[Lom06] Andrea Lombardoni. *Towards a Universal Information Platform: An Object-Oriented, Multi-User, Information Store*. PhD thesis, ETH Zurich, Switzerland, 2006. doi:10.3929/ethz-a-005364615.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.*, 37:316–344, 2005. doi:10.1145/1118890.1118892.

[MSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing Adaptive Software. *Computer*, 37(7):56–64, 2004. doi:10.1109/MC.2004.48.

[NGD+08] Moira C. Norrie, Michael Grossniklaus, Corsin Decurtins, Alexandre de Spindler, Andrei Vancea, and Stefania Leone. Semantic Data Management for db4o. In *Proc. Intl. Conf. on Object Databases (ICOODB)*, pages 21–38, 2008.

[NGLN10] Michael Nebeling, Michael Grossniklaus, Stefania Leone, and Moira C. Norrie. Domain-Specific Language for Context-Aware Web Applications. In *Proc. Intl. Conf. on Web Information Systems Engineering (WISE)*, pages 471–479, 2010. doi:10.1007/978-3-642-17616-6_42.

[NGLN12] Michael Nebeling, Michael Grossniklaus, Stefania Leone, and Moira C. Norrie. XCML: Providing Context-Aware Language Extensions for the Specification of Multi-Device Web Applications. *World Wide Web Journal*, pages 447–481, 2012. doi:10.1007/s11280-011-0152-2.

[NNNH04] Dag Nyström, Mikael Nolin, Christer Norström, and Jörgen Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proc. Workshop on Software Engineering for Automotive Systems*, pages 1–8, 2004.

[Nor93] Moira C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems. In *Proc. Intl. Conf. on the Entity-Relationship Approach (ER)*, pages 390–401, 1993. doi:10.1007/BFb0024382.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, 2004.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 1099–1110, 2008. doi:10.1145/1376616.1376726.

[OSV11] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima, Inc., 2nd edition, 2011.

[PBJ98] František Plášil, Dušan Bálek, and Radovan Janeček. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proc. Intl. Conf. on Configurable Distributed Systems (CDS)*, pages 43–51, 1998. doi:10.1109/CDS.1998.675757.

[RSS+08] Marko Rosenmüller, Norbert Siegmund, Horst Schirmeier, Julio Sincero, Sven Apel, Thomas Leich, Olaf Spinczyk, and Gunter Saake.

FAME-DBMS: Tailor-Made Data Management Solutions for Embedded Systems. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 1–6, 2008. doi:10.1145/1385486.1385488.

[SC05] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 2–11, 2005. doi:10.1109/ICDE.2005.1.

[SM03] Seyed Masoud Sadjadi and Philip K. McKinley. A Survey of Adaptive Middleware. Technical Report MSU-CSE-03-35, Department of Computer Science, Michigan State University, December 2003.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1150–1160, 2007.

[SSH+05] Junrong Shen, Xi Sun, Gang Huang, Wenpin Jiao, Yanchun Sun, and Hong Mei. Towards a Unified Formal Model for Supporting Mechanisms of Dynamic Component Update. *SIGSOFT Software Engineering Notes*, 30(5):80–89, 2005. doi:10.1145/1095430.1081720.

[Ste98] Andreas Steiner. *A Generalisation Approach to Temporal Data Models and Their Implementations*. PhD thesis, ETH Zurich, Switzerland, 1998. doi:10.3929/ethz-a-001923958.

[SZ09] Daniel Spiewak and Tian Zhao. ScalaQL: Language-Integrated Database Queries for Scala. In *Proc. Intl. Conf. on Software Language Engineering (SLE)*, pages 154–163, 2009. doi:10.1007/978-3-642-12107-4_12.

[SZD07] Ionut Emanuel Subasu, Patrick Ziegler, and Klaus R. Dittrich. Towards Service-Based Data Management Systems. In *Proc. Workshop on Database Systems in Business, Technology and Web (BTW)*, pages 296–306, 2007.

[SZDG08] Ionut Emanuel Subasu, Patrick Ziegler, Klaus R. Dittrich, and Harald Gall. Architectural Concerns for Flexible Data Management. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 34–39, 2008. doi:10.1145/1385486.1385497.

[TB06] Wee Hyong Tok and Stephane Bressan. DBNet: A Service-Oriented Database Architecture. In *Proc. Intl. Conf. on Database and Expert Systems Applications (DEXA)*, pages 727–731, 2006. doi:10.1109/DEXA.2006.48.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[XOW04] Bo Xu, Aris Ouksel, and Ouri Wolfson. Opportunistic Resource Exchange in Inter-Vehicle Ad-Hoc Networks. In *Proc. Intl. Conf. on Mobile Data Management (MDM)*, pages 4–12, 2004. doi:10.1109/MDM.2004.1263038.

About the authors

Michael Grossniklaus is an assistant professor for databases and information systems at the University of Konstanz. His research focuses on novel database techniques for emerging application domains, such as context-aware data management, data stream processing, or graph data. Currently, he is investigating how object database technologies can be used for cloud data management. Contact him at `michael.grossniklaus@uni-konstanz.de`.

Stefania Leone is a post-doctoral researcher in the Semantic Information Research Laboratory at University of Southern California. Her research interests are in the field of information systems and object databases, both at the conceptual and the technological level, focusing on new ways to enhance and facilitate their design and development process. Contact her at `stefania.leone@usc.edu`.

Alexandre de Spindler is a lecturer of information systems at the Zurich University of Applied Sciences. His research interest include the design and development of domain-specific support for information systems development. He is currently investigating the requirements of a system able to generate domain-specific web information systems. Contact him at `alexandre.despindler@zhaw.ch`.

Moira C. Norrie is a full professor and head of the Global Information Systems group at ETH Zurich. Her research and teaching focusses on the use of object-oriented and web technologies for next generation information systems. Her research group developed the OMS database development suite designed to support the development of object databases from conceptual design through to implementation. Contact her at `norrie@inf.ethz.ch`.

Acknowledgments Michael Grossniklaus' work is partially funded by the Swiss National Science Foundation (SNF) grant PA00P2_131452. The authors would like to thank Michael Nebeling who provided the implementation of the XCM metamodel extension module, Matthias Geel who implemented the information component module as well as Christoph Lins and Julian Sgier who contributed the metamodel extension module for events.