Interoperable Systems Group
Dublin City University
Glasnevin, Dublin 9, IRELAND.

Title: Storage of Complex Business Rules in Object Database

Author: Dalen Kambur

Project: EGTv

REF: ISG-02-09

Date: August 05, 2002

Abstract

True integration of large systems requires sharing of information stored in databases beyond sharing of *pure data*: business rules associated with this data must be shared also. This research focuses on providing a mechanism for defining, storing and sharing business rules across different information systems, in an area where existing technologies are weak. In this paper, we present the *pre-integration* stage where individual business rules are stored in the database for subsequent exchange applications and information systems.

1 Introduction, background and motivation

Federated database systems (FDS) integrate different, mutually incompatible *component* database systems [SL90]. Client application requests are issued against the FDS, which applies them against local databases. The FDS model is particularly useful in multimedia databases, where component databases store multimedia data using different multimedia data types, each of which defines different *operations* on the stored data. In a FDS these different data types must have an identical access interface, which requires reusing existing operations and definition of new operations in the FDS. These operations are a form of *business rules*, transformation patterns associated with the data, rather then any particular application.

One multimedia system that could benefit from federated multimedia database architecture is the *Físhlár Digital Multimedia System* [LSO+00]. This system uses Oracle Video Server software in a single server environment. As this can be scaled only to a certain (technological) limit, the processing power and the available disk space are limited, thus limiting the number of concurrent users and the amount of digital multimedia material that can be stored with the system. Distribution of processing and data across several servers is needed, where individual servers can be based upon mutually incompatible technologies. Thus, a federated multimedia system is suitable in this case. Business rule sharing is essential as it facilitates incorporation of mutually incompatible technologies, and can help with performance by distributing the work-load against multiple servers.

This research does not deal with actual algorithms used to extract, stream or modify multimedia content, rather it provides an infrastructure for developing multimedia database federations. With multimedia databases, an essential part of the data stored with the database are manipulation rules, as different types of multimedia databases might store and manipulate multimedia content in a different manner. These multimedia manipulation rules represent a form of complex business rule*s*. The Físhlár system for example, is intended only for streaming digital multimedia content to multiple users. The streaming of multimedia material itself is a complex business rule; involving transformation of the stored multimedia content into series of *chunks* each of which can be independently transmitted to a client application (for instance, web browser) and displayed. Another complex rule might add a special effect in a specified place in existing multimedia material, involving searching through the multimedia material for the place to insert the special effect, inserting the special effect and storing the end result with the database. In both of these cases, the business rule is too complex to be defined in a *database specific programming language* (DBPL); and the processing should be carried out by the database server in order to maximise performance.

The currently available object database technologies, object-oriented databases based upon ODMG [CB99] and object-relational (O-R) SQL1999 [GP99] standards are deficient in this respect: whereas ODMG does not identify a need to define or share business rules, SQL1999 offers only a DBPL for defining business rules. Using CORBA [HV99], business rules can be defined and shared for distributed objects, and its Persistent Storage Services [Ion02] enable its usage with persistent objects stored in different types of databases. Usage of PSS however, is a highly complex low-level and error-prone development process which *shifts* the attention of a developer from the system to be built to low-level technical details.

Related research projects are generally limited through propriety of their data model, communication protocol and lack of published metamodel. The contribution in our research project is the provision of sharing of business rules for object databases using standard technologies to overcome compatibility issues. This builds upon the work of-

fered in [RB02].

This paper is structured as follows: In §2 an overview of related research projects is given. In §3 we examine the categories of business rules, where for each category is identified how can it be used in object-oriented modelling. §4 introduces the architecture we use in this research with some operational scenarios. §5 contains important implementation details with an overview of used technologies. Finally, conclusions are given in §6.

2 Related Research

In this section, we review how existing projects share business rules. For each project, we explain how the business rules are modelled, what the limitations of the project are, and integration capabilities with regard to business rules. We conclude the description of each project with those aspects that are employed in our research.

2.1 The MOOD Project

In the METU Object-Oriented DBMS (MOOD) project [DAO+95] , the business rules are modelled as C++ methods and compiled to dynamic libraries. To apply a business rule against the data, a client application specifies the rule to be applied and parameters, where the DBMS loads the library containing the rule and applies it against the data. This communication is based on a proprietary protocol, details of which were never published making it difficult to integrate a MOOD database with any other database. Business rules cannot be invoked within queries, which significantly reduces power of queries to built-in operations. *Example 1* is invalid because the `WHERE` clause of the query applies business rule `contains` to identify whether the description of a program contains `Cherry`.

Example 1

```
SELECT name FROM Program
WHERE desc.contains("Cherry")
```

An approach to compile business rules into individual libraries may introduce an overhead with business rules defined using other business rules, i.e. if business rule A is defined using business rules B and C, then prior to application of business rule A, libraries containing business rules B and C must be loaded. We consider this approach to have too fine level of granularity.

MIND (METU INteroperable DBMS) is the integration architecture used in this research. In this architecture, individual databases are wrapped using CORBA objects, thus employing a standard protocol. However, the integration architecture does not facilitate the application of business rules on integrated database systems even if the integrated database is a MOOD database. Furthermore, no standard metamodel has been published as part of this project, which is an important prerequisite for efficient database integration. However, the general approach chosen in this project, i.e. to use a general purpose programming language to define complex business rules and to compile them into dynamic libraries is well suited to our own approach.

2.2 The LOQIS Database System

In the LOQIS database system [SBMS94], business rules are modelled using the Stack Based Query Language (SBQL) as *procedures* and stored with the database. A procedure [SKL95] consists of a procedure name, parameters and body, where the name is an identifier, parameters are *expressions* (essentially, queries) and the body consists of expressions and control statements such as `if..then..else` and they return an expression. In *Example 2*, taken from [SKL95] procedure `ChangeDept` changes the department the employees work in for all employees listed in `E`. `E` can be a single employee, a set of employees or any kind of expression (i.e. query), providing a simple and powerful way to define a procedure.

Example 2

```
procedure ChangeDept(var E; var D)
  begin
   delete (DEPT.EMPLOYS) where EMP∈E;
   (e←E).(
    create local EMPLOYS(↑e);
    D⊴=EMPLOYS;e.WORKS_IN:=↑D;
    e.EDNO:=D.DNO;)
  end ChangeDept;
```

For integration LOQIS uses *views*, where a view is a result of procedure execution, thus facilitating powerful transformations on stored data. LOQIS

views can also include behaviour. The project does not directly address integration of different data sources. Only parts of the metadata side of the project were published. SBQL, though powerful and clearly defined, is rather abstract and close to the art of programming languages. However, the concepts introduced in SBQL are built into our approach, where we provide a more traditional external interface.

2.3 The MultiView Project

In the MultiView project [HR93] business rules are defined as methods written in Opal and stored in the GemStone database, where Opal is GemStone's version of the SmallTalk. Opal employs total encapsulation (i.e. data is accessible only via accessor methods), and used to emulate multiple inheritance, originally not provided with the GemStone. A technique named *object-slicing* is employed and facilitates non-consecutive storage of properties of an object. This retains a single object [RKR94] representation to methods, thus being ideal for representing virtual objects. However, the implementation of the technique in the MultiView is restricted to SmallTalk which is not widely spread programming language.

With regards to integration, the project focuses on the integration of other MultiView based databases and resolving structural differences. Interoperability with other types of databases was not part of the research. For integration, views are provided as sets of virtual classes, each of which is defined using integration operations over other virtual or base classes. The operations provided are simple, setting an accent on set operations on objects (`select`, `union`, `intersect` and `diff`). The `refine` operation is used for redefining associations, changing types of properties and similar purposes resulting in adding new properties. The `hide` operation results in removing existing properties. Redefinition of inheritance of virtual classes is impossible with the provided operations. As the model employs total encapsulation, these properties are business rules defined in Opal using other properties of the virtual class to be defined, and the base classes. For representing virtual objects, object-slicing is used, which is an aspect of this research project that will take part in our research for representing virtual objects. There are no details of a metamodel specification.

2.4 The IRO-DB Project

In the IRO-DB project [BFN94] business rules are provided for users of the federation of relational and object-oriented databases. Rules are defined using the C++ programming language, following the Object Manipulation Language (OML) definition from ODMG standard, as part of the interface to virtual class. IRO-DB does not facilitate an access to the properties directly, instead accessor operations are generated. The rules are compiled and made available to the federated applications and queries. No discussion as to what business rules are allowed to do (only to query or alter data) and the outcome of modifying operations (as we discuss in §3) is offered. In this research, an early draft of the standard ODMG metamodel was used. The communication protocol used between client application and the server is modified Call Level Interface (CLI), however in the later research they have used CORBA instead [RBPF98], by exporting the objects and business rules using CORBA interface. In our research CORBA is used in similar fashion, where we facilitate definition and usage of business rules at preintegration stage and in the future of the research, in federated database.

3 A Taxonomy of Business Rule Formats

Client applications operate on data stored in databases in different fashions: some of the operations applied against data are client application specific, whereas other operations are more associated with the data itself. Such operations are *business rules* and need to be stored with the database so that more then one client application can use them. In this section a detailed taxonomy of operations is presented with an explanation as to how each particular format of operation is supported in our architecture. Operations can be distinguished by

- their return value;
- if they modify objects (modifiability);
- the parameters they receive.

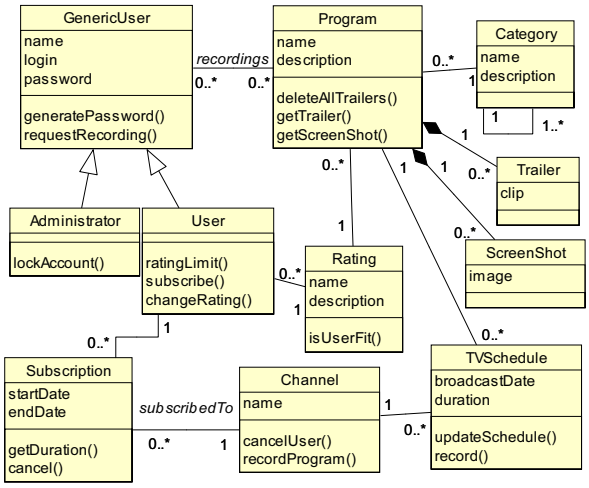

Figure 1: The Recorder schema.

The categories are first introduced (§3.1) and later, combinations of the categories are discussed and explained (§3.2). In the remainder of the article a multimedia database schema illustrated in *figure 1* will be used. The schema belongs to a digital multimedia provider which provides its customers with capability to record multimedia material broadcast on different channels and later on to access to that material. User subscribes to a particular channel, and the same program can be scheduled more then once, on different channels and broadcasting dates.

3.1 Rule Categories

- Return value

Operations can either return a value or not. A return value is calculated upon invocation, not stored with the database, and can be used in expressions as a simple value. The return value can be of *primitive type* or an *object reference*. While a primitive type can be easily passed into an operation, an object reference must maintain the location reference and thus requires additional handling.

Operations returning no value are similar to *procedures* and can be used to modify the state of an object or to query and return the state of the object via output parameters. Operations returning an object reference allow client applications to get references and manipulate stored objects.

- Modifiability

In order to explain some of the issues involved, we introduce two terms to refer to objects modified during operations. The object of which class the operation is defined is known as the *target object*. An object connected (through a relationship property) to the target object, and which is also affected by the original operation, is referred to as the *related object*. Write operations create, update or delete at least one object. An update operation that changes only the target object is considered a *level 0 update* operation. In other words, it does not affect another object indirectly.

Operations may update related objects. We refer to these changes as *level 1 changes*. For instance, an operation `updateSchedule(program)` of class `Channel` updates the time the program is broadcast by modifying the `broadcastDate` property of a related object of class `TVSchedule`.

Write operations can also create and delete related objects. This feature is based upon an object-factory design pattern [GHJV98] where an object-factory class is responsible for providing an interface for creating and deleting the objects of the managed class. These changes are also categorised as *level 1 updates*. For example, an operation `deleteAllTrailers` of the class `Program` does not modify the target object, but it deletes related objects of class `Trailer` (the cancelled broadcasts of the program).

Similar to level 1 updates, level n updates include any changes made to related objects of the related objects, where n is the number of relationships between the modified object and the target object.

Invocation of write operations in queries involves an additional risk as the order of evaluation is not guaranteed in query languages and thus might result in different results which is discussed later on.

- Parameters

An operation may contain either zero or more parameters. Each parameter can be of *primitive type* or a *reference*. Also, each parameter can be either *read* or *write*. In the case where the operation directly manipulates an object, its structure and interface must be known at compile time, where the interface of primitive type is always known at compilation time (primitive type is atomic), i.e. does

not have structure. For example, a business rule `subscribe(channelName)` of class `User` must know the structure of the `Channel` class to identify the `channel` object which the `User` wants to subscribe to.

However, in the case where the operation delegates the parameter manipulation to an operation of some other class and does not manipulate the parameter itself, the structure and interface of the parameter need not be known during compilation. For example, a business rule `recordProgram(user,date)` of class `Channel` might use the `record(user)` rule of class `TVSchedule` that designates a program to be recorded for the specified `user`, thus not manipulating `user` directly.

3.2 Category Combinations

In this subsection, we shall classify each operation based on its format into A, B or C category, where category A operations preserve an object's state, category B operations modify an object's state, and finally, category C operations are not supported. An overview of formats of operations and their corresponding categories is given in *table 1.* This categorisation is useful to determine the extent to which an operation can be supported with our architecture, which is explained further on.

Operations that preserve both an object's state and parameters are classified as category A and can be used in queries and in client applications. For instance, operation of format 3 returns a value of an expression based on the object's state preserving an object's state and parameters, thus, if used in a query, the evaluation order will not affect the results and it is safe to use such an operation in queries.

Operations that change an object's state or operation parameters (classified as category B) can be used from client applications, but cannot be safely used in queries. Changes within the operation invocation and their order of execution might affect the result of the query. Changes are supported (level 0 and higher) where the developer must assure that the changes are not conflicting. In general, if changes caused by invoked operations do not interfere with the selection criterion, and if the operations modify different objects the query is safe to execute. Clearly, the safeness of an operation execution depends not only on an operation, but the query as well. For instance,

Example 3

```
SELECT s.user.Name
FROM (SELECT s1.*
FROM Subscription s1
WHERE s1.subscribedTo.name
    = "SuperTV Plus") s
WHERE s.cancel() > 30;
```

is to select the names of all the subscribers of the SuperTV Plus channel which need to be refunded (have more then 30 days of subscription left unused) because the channel cancelled subscription with the provider. Same operation in a similar query is not safe, as illustrated in *example 4*.

Example 4

```
SELECT s.user.Name
FROM Subscription s
WHERE s.cancel() > 30 and
  s.subscribedTo.name
    ="SuperTV Plus";
```

If `cancel` operation is executed before filtering against the `Channel`, the query would cancel subscriptions to all channels and return the names of users that need to be refunded of SuperTV Plus channel. The `cancel` operation is of format 4 and as it can affect query execution, belongs to category B.

Operations not retrieving or making changes to an object are not used in modelling object-oriented systems and in the *table 1* are classified to be of category C. For instance, operations of format 1 make no sense as they make no changes to any object nor they retrieve a value.

Though it might seem overly simple, this is the maximum extent to which an operation defined using a standard programming language can be classified. To determine if an operation is safe in a query, a formal description of both the operation and the query would be required, which is not available with standard programming languages and standard query languages.

- Rule Formats

Operations returning no values and modifying no objects have no use in any application and are not supported.

Format	Form	Modify	Category
1	$f()$		C
2	$f()$	$\surd$	B
3	$x = f()$		A
4	$x = f()$	$\surd$	B
5	$f(r_1, ..., r_n)$		C
6	$f(r_1, ..., r_n, w_1, ..., w_m)$		B
7	$f(r_1, ..., r_n)$	$\surd$	B
8	$f(r_1, ..., r_n, w_1, ..., w_m)$	$\surd$	B
9	$x = f(r_1, ..., r_n)$		A
10	$x = f(r_1, ..., r_n, w_1, ..., w_m)$		B
11	$x = f(r_1, ..., r_n)$	$\surd$	B
12	$x = f(r_1, ..., r_n, w_1, ..., w_m)$	$\surd$	B

Table 1: Combinations of Categories.

These operations are used to change an object's state, are not dependent on external parameters and generate no return value. For example, an operation `deleteAllTrailors()` of the class `Program` deletes all the trailors of the target program.

These operations retrieve an attribute's value or use an expression to derive a new value, while preserving the object's state. For example, the `getDuration()` operation of the class `Subscription` returns a duration of the subscription in days (i.e. `endDate - startDate`).

This type of operation is permitted to modify the objects and also generates a return value. For example, operation `cancel()` of class `Subscription` cancels the subscription the user had to a particular channel. The previous value of `endDate` attribute is returned to facilitate potential financial compensation.

For reasons similar to format 1 operations, these operations are not supported.

These operations allow modification to parameters only. The object's state is preserved and no value is returned. For example, an operation `getTrailer(client)` of the `Program` class streams to the client the trailer (and thus, modifies the `client` object). In this, the state of the `Program` object is not modified.

These operations are used to modify objects. They receive non-modifiable parameters and generate no return value. For example, an operation `cancelUser(user)` of `Channel` class canlcels the subscription to the channel for the `user`.

These operations are permitted to modify both an object's state and parameters, while generating no return value. For example, the `subscribe(channel)` operation of the class `User` subscribes an user to the `Channel`. This operation creates an object of class `Subscription`, and associates it with the referenced `channel` object and `User` object, thus modifying both of them.

These operations generate a return value while preserving the object's state. For example, an operation `isUserFit(user)` of the class `Rating` returns `true` if the user is is fit to watch programs of that rating.

These operations alter some of the parameters they receive and generate a return value while preserving the object's state. For example, an operation `getScreenShot(index,image)` of the class `Program` places the screenshot number `index` in the parameter `image` and returns the number of remaining images.

These operations change an object's state and generate a return value based on the non-modifying parameters they receive. For example, an operation `requestRecording(program,channel,time)` of the class `User` requests recording of a particular program on a specified channel by creating an association between `GenericUser` and `Program`.

These operations change an object's state and input parameters they receive, while also generating a return value. For example, an operation `changeRating(newRating)` operation of the class `User` changes the rating of the movies the user is allowed to watch, returning the reference to the previous rating of the user.

4 Architecture and Operational Scenarios

The architecture presented here provides developers with the ability to store business rules inside database and remote client applications with access to them. It is an extension of [KR01] in which only ODMG databases were supported. This architecture facilitates both O-R and ODMG databases and is introduced through scenarios that cover different operational aspects of the architecture: registering classes for adding new business rules (§4.1); reusing existing business rules (§4.2); and specifying new

business rules (§4.3). The ODL_b language used in these scenarios is introduced through a series of examples with a full formal specification given in [Kam02].

In this architecture, the classes must be registered with the system before business rules can be reused or defined. The registration process requires the classes from the database schema to be specified and subsequently, existing or newly defined operations are available to all client applications or query engine. In this section, we illustrate this process of making classes and operations available to all client applications.

4.1 Registering classes for adding new business rules

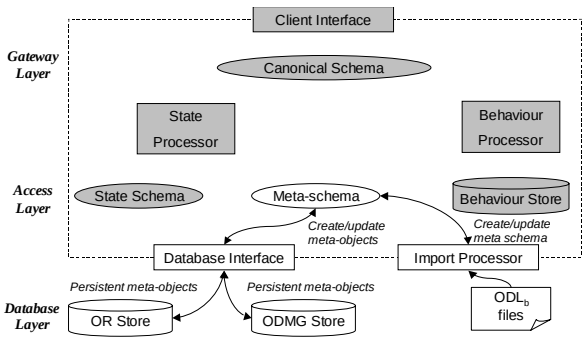

Figure 2: Registering a class.

In the scenario illustrated in *figure 2*, classes for which operations are stored and shared are made available to the system. The database is located at the `Database Layer`, and can be an ODMG or an O-R database. The `Import Processor` reads the ODL_b `file` containing a list of classes to be registered. This list is used to create (with the incorporation of a new database) or update (with the incorporation of new classes) the `Meta-schema`. The `Meta-schema` is a canonical form of the database metadata and is maintained by the `Database Interface` which transforms native database objects to their canonical form (to be used by other layers). The information stored in the `Meta-schema` is subsequently used to execute client application requests. The shaded parts of the architecture in *figure* 2 are not used in this scenario and will be described later.

The ODL_b language features are introduced in the remainder of this subsection.

Example 5

```
database MultimediaProvider {
  import GenericUser;
}
```

In *Example 5*, only the class `GenericUser` is imported. The `database` declaration specifies the name of the database the class will be taken from. The `import` declaration specifies the names of the classes to be registered. In this case, the relationship to `Recording` will not be imported as it has no meaning. Please refer to §4.2 for the affect of these commands to business rules.

A more comprehensive way to register larger and more complex schemas is provided using the optional `sub`, `rel` and `all` keywords specified after the class name. The `sub` will include all subclasses of the specified class; the `rel` will register all classes connected to the specified class using relationships; and the `all` keyword combines both of these.

Example 6

```
database MultimediaProvider {
  import GenericUser sub;
}
```

In *Example 6*, all the `GenericUser` subclasses are registered (i.e. `Administrator` and `User`). As classes `Program`, `Rating` and `Subscription` are not registered, the relationships to these classes are dropped.

Example 7

```
database MultimediaProvider {
  import GenericUser rel;
}
```

In *Example 7*, all classes except of class `Administrator` are registered as all of them are accessible via relationships.

Example 8

```
database MultimediaProvider {
  import GenericPerson all
  exclude Rating;
}
```

In *Example 8*, it is shown that a limitation can be placed on set of classes to be registered, namely, class `Rating` is excluded along with relationships from `User` and `Program` to this class. Other classes and relationships are registered. Please refer to [Kam02] for a more detailed description of language and examples.

4.2 Using business rules from existing applications

O-R and ODMG databases differ in their support for storage of operations. ODMG databases presume all operations are part of client applications, and the database server contains only an object's state. Such operations, as part of the client application, cannot be extracted and reused. However, source modules can be used for redefining the operations as "new", as explained in §4.3. O-R databases facilitate a limited storage of operations. These operations are defined using a proprietary database programming language[1] (such as Oracle PL/SQL). Part of the database server's functionality is to implement such operations. Thus, these operations are not part of the client application (they are already separated) and can be reused. However these operations cannot be directly imported. Instead operation names and signatures (i.e. parameter types and return value) are stored in the meta-schema. Client application requests for invocation of the reused operation are forwarded to the `Database Layer` and executed by the database server. Results of the execution are also retrieved from the database server.

A scenario of importing an existing operation is illustrated in *figure 2* (§4.1), where the `Import Processor` reads names and signatures of all available operations for each class to be incorporated from the `Database Layer`. The signatures are transformed to a common form which is also used to specify new operations and stored in the `Meta-schema`. The stored information is propagated to the `Database Layer` via the `Database Interface` and is subsequently used by the `Behaviour Processor` to invoke operation.

The commands in §4.1 are used to import O-R behaviour together with class definitions. In *Example 5*, the `generatePassword()` and `requestRecording()` business rules will be imported, as only class `GenericUser` was imported. In *Example 6*, business rules `generatePassword()`, `requestRecording()`, `lockAccount()`, `ratingLimit()`, `subscribe()` and `changeRating()` will be imported. In *Example 7*, business rules `generatePassword()`, `requestRecording()`, `lockAccount()`, `ratingLimit()`, `subscribe()`, `changeRating()`, `cancelUser()`, `deleteAllTrailers()`, `getTrailer()`, `getScreenShot()`, `isUserFit()`, `record()`, `updateSchedule()`, `recordProgram()`, `getDuration()` and `cancel()` will be imported. In *Example 8*, business rules `generatePassword()`, `requestRecording()`, `ratingLimit()`, `cancel()`, `subscribe()`, `changeRating()`, `cancelUser()`, `deleteAllTrailers()`, `getTrailer()`, `record()`, `updateSchedule()`, `recordProgram()`, `getDuration()` and `lockAccount()` are imported.

Before our new architecture for object database systems, none of these rules were available to client applications except as part of proprietary applications. These rules are now available to all client applications.

4.3 Specifying new business rules

The specification of new business rules is illustrated in *figure 3*. Following traditional programming languages, this is a two stage process in which the business rule is first *declared* and then *defined*[2]. To declare a new business rule, the `Import Processor` reads ODL_b `files` containing their declarations. These declarations contain a list of operations and signatures, module specifications and files for building the modules, are stored with the `Meta-schema`, and forwarded to the `Database Layer` using the `Database Interface`.

In *figure 3* the process of defining operation is presented. An operation is defined similarly to a method in a programming language, i.e. using object attributes and other, already defined, methods. For this, an extended form of programming language is used which facilitates using ODL_b declarations instead of programming language declarations. The `Import Processor` reads files spec-

[1] Though the O-R database standard defines a programming language, we are unaware of any compliant O-R database.

[2] The broken line linking the `Import Processor` to the `Behaviour Store` takes place only when defining business rules.

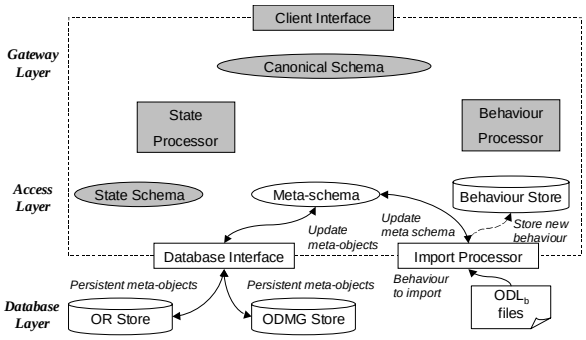

Figure 3: Specifying new behaviour.

ified as belonging to a single module and builds a single, dynamically loadable library by translating ODL_b declarations to programming language declarations. Modules are stored with the `Behaviour Storage`, and registered with the `Meta-schema`, thus making newly defined methods available for invocation. The `Behaviour Storage` is a register of available modules and it is implemented as a folder on a file system.

5 Implementation

In this research only standard technologies, available on variety of platforms and in different programming languages are used. Our architecture requires a technology that facilitates data (parameters, object references and return values) transfer between different platforms, so we have chosen CORBA (Orbix).

It also requires a number of programming and query language processing tools, and all of them are designed using ANTLR [ANT02] which generates parser classes from the specification file containing rules (encoded in a customised version of Extended Backus-Naur Form), followed by *semantic action*s. The parser invokes the semantic actions for each construction recognised in source files.

The Versant Developer Suite (VDS) is an ODMG database used for the first prototype. Our platform is Windows based, though all of the technologies we used should be easily portable to other platforms. Microsoft Visual C++ compiler was used to operate with the VDS and Orbix. A Linux version is under development.

The prototype is built as a standard client-server system, where server is used by the client applications using a `Client Interface`, packaged in `Database` and `Session` classes. For each database, a single instance of the `Database` class is registered with the CORBA Naming Service and acts as an object factory for the `Session` class. The `Session` class provides a client application interface to both `Object Processor` and `Query Processor` in the form of separate methods. To establish a connection, client applications invoke the `getSession()` method of the `Database` object registered with the CORBA Naming Service using the database name.

For each persistent class, a wrapper with the same name is (re)generated as described in the operational scenarios §4.1, §4.2 and §4.3. The wrapper has a set of operations bound to the operations of persistent classes which package the operation invocation request. The wrappers also have the same set of properties as the persistent classes which are synchronised with the persistent objects where object-slicing [HR93] is used to assure forwarding of changes to the server.

Similarly, wrappers are generated for server-side operation implementation. They propagate changes to the database and provide hooks to which client application requests are forwarded, this way implementing parts of the `State Processor` and `Behaviour Processor`. Object-slicing is also used here to forward the changes to the database. However, this implementation of object-slicing will be reused for subsequent integration phase.

6 Conclusions

In this paper our approach to storing and accessing business rules in object databases was demonstrated. We have shown the importance of this facility in one of the most prominent areas of information technology applications in business: multimedia databases. Despite the importance of sharing business rules, the current industry standards provide very little support. We presented our classification of business rules based on their format, and the scenarios in which they are used (i.e. query as opposed to client application). This classification also used to describe the extent to which each particular business rule can be supported in our architecture. Rather then using a full formal de-

scription, we provided the reader with a more informal series of scenarios, illustrated with examples. Nevertheless, we provided implementation details (technologies used and overall design) to give an overview of the technical background of this work. In the future research we will focus on extending the architecture to provide business rules in object-oriented views, which is the ultimate aim of this research. In the introduction section of this paper, we briefly mentioned the application area of object-oriented views: multimedia systems and business systems integration. Bearing in mind the broad architectural design, we feel such an extension will be simple, requiring more theoretical research related to updatability of, and inclusion of, behaviour in object-oriented views.

References

[ANT02] ANTLR, *ANTLR Reference Manual*, ANTLR, 2002.

[BFN94] Busse, R., Fankhauser, P. and Neuhold, E. J., Federated Schemata in ODMG, in *East/West Database Workshop*, pp. 356–379, 1994, URL `citeseer.nj.nec.com/225897.html`.

[CB99] Catell, R. and Barry, D., *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 1999.

[DAO+95] Dogac, A. et al., METU Object-Oriented DBMS Kernel, in *Proceedings of the 6th International Conference on Database and Expert Systems Applications*, pp. 14–27, Springer Verlag, 1995.

[GHJV98] Gamma, E. et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1998.

[GP99] Gulutzan, P. and Pelzer, T., *SQL-99 Complete, Really*, R&D Books, 1999.

[HR93] Harumi, K. and Rundensteiner, E., Developing an Object-Oriented View Management System, in *Proceedings of the Centre for Advanced Studies Conference (CASCON)*, pp. 548–562, 1993.

[HV99] Henning, M. and Vinoski, S., *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.

[Ion02] Iona Technologies Plc, *CORBA Programmer's Guide C++*, Iona Technologies Plc., 2002.

[Kam02] Kambur, D., Storage of Behaviour in Object Databases, in *Technical Report ISG-02-02*, pp. 1–13, 2002.

[KR01] Kambur, D. and Roantree, M., Using stored behaviour in object-oriented databases, in *Proceedings of the 4th Workshop EFIS 2001*, pp. 61–69, IOS Press, 2001.

[LSO+00] Lee, H. et al., The Físchlár Digital Video Recording, Analysis, and Browsing System, in *Proceedings of the RIAO 2000 - Content-based Multimedia Information Access*, 2000.

[RB02] Roantree, M. and Bećarević, D., Metadata Usage in Multimedia Federations, in *First International Meta informatics Symposium (MIS 2002)*, 2002.

[RBPF98] Ramfos, A. et al., CORBA Based Data Integration Framework, in *Proceedings of the Third International Conference on Integrated Design and Process Technology, IDPT*, pp. 176–183, 1998.

[RKR94] Ra, Y., Kuno, H. and Rundensteiner, E., A Flexible Object-Oriented Database Model and Implementation for Capacity Augmenting Views, *Techical report*, Department of Electronic Engineering and Computer Science, University of Michigan, 1994.

[SBMS94] Subieta, K. et al., A Stack-Based Approach to Query Lanugages, in *Proceedings of the Second International East/West Workshop*, pp. 159–180, Springer Verlag, 1994.

[SKL95] Subieta, K., Kambayashi, Y. and Leszczyłowski, J., Procedures in Object-Oriented Query Languages, in *Proceedings of the 21st International*

Conference on Very Large Data Bases, pp. 182–193, Morgan Kaufmann, 1995.

[SL90] Sheth, A. and Larson, J., Federated Database Systems for Managing Distributed, heterogeneous and Autonomous Databases, in *ACM Computing Surveys*, vol. 22(3), pp. 183–226, 1990.