

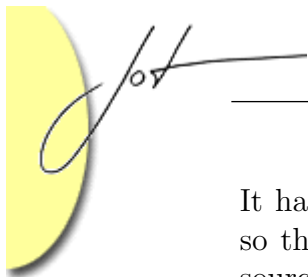
## An Object Model for Sensor Data Integration

**Dalen Kambur**, School of Computing, DCU, Glasnevin, Dublin, Ireland  
**Mark Roantree**  
**John Murphy**

One example of large volumes of distributed data is that of sensor networks, where dedicated sensing equipment is used to monitor events and happenings in a wide range of domains, including monitoring human biometrics. In areas such as Personalised Health, large volumes of data are generated to provide an early indication of potential health problems. Our research is focused on sensor data that has been enriched in the form of XML trees, so that each sensor device can be regarded as an XML tree, and in some cases, containing services. While this architecture offers a certain degree of interoperability, current Web Services approaches are not suited to the difficult tasks of integration, restructuring of information and global issues such as updates. While a Web Services approach offers the advantage of loose coupling of data sources, a more formal model is required to underpin the sensor network. In this paper we describe our Object-Reference model (ORef), which extends a standard object-oriented model with *references* as the sole concept for locating objects (sensor objects) and their properties, and where the basic object-oriented modelling primitives are respecified to incorporate references. References are also used to define *transformations* present in the model in the sense that they are *closed*, meaning that each transformation operates on one or more objects, and results in one or more objects. The purpose of this model is to provide a stable bedrock for the loose coupling of data sources and services in an XML sensor network.

### 1 INTRODUCTION

The levels of interest in pervasive computing and ubiquitous sensing are significant enough to see the development and deployment of sensing technology all around us. One can also see the emergence of applications such as environmental monitoring and ambient assisted living which leverage the data gathered and present us with applications that seek to improve our daily lives. However, most of the developments in this area have been concerned with either developing the sensing technologies, or the middleware to gather this data, and the issues have included power consumption on the devices, security of data transmission, networking challenges in gathering and storing the data and fault tolerance in the event of network or device failure. If we assume that these issues will be addressed successfully in the short term, we are still required to develop applications that are robust and flexible, and at such time the issues of high-level querying and updating of global data becomes a major issue.



It has been shown in previous work [LRJ<sup>+</sup>07] that sensor devices can be enriched so that their output is captured into XML documents or databases. These XML sources may also have services associated with them.

The problem we address in this paper is how to properly integrate these sources of data so that we can query them at a global level, restructure the data to meet the needs of heterogeneous applications, and provide updating capabilities where sensor data output requires recalibration or normalisation.

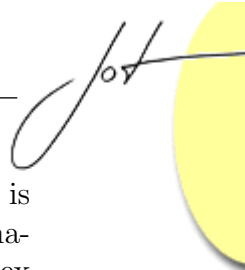
## Motivation and Contribution

In our application of sensor networks, we assume that sensor devices are heterogeneous, and that the generated data is stored with the database and exposed using a Web Service in an XML format. In general, a single sensor device offers little meaningful information and requires the collaboration with a number of other devices to generate meaningful output. The problem is that different applications will require different collaborations and thus, we cannot form groups of sensors in advance. In effect, this means that we must continually integrate sensor data sources during the lifetime of the network. This presents a more traditional database integration problem similar to federated database framework [SL90]. A modern approach to this problem is to use a loosely coupled Web Services architecture to enable collaboration. However, such an approach: (1) is only a top-level integration solution, and (2) does not address any specifics of individual Web Service implementation in terms of data model, behaviour and collaboration. Our contribution to this problem is to provide a new object model that: (1) is respecified to allow a precise definition of object-oriented database modelling features, such as user-defined types, operations and database queries, and (2) is used as the platform for providing Web Services. In the ORef model, any transformation remains closed and does not require an extension of the model primitives. Results of a transformation remain linked to their sources which allows the updates of the transformation results to be propagated to their sources. Thus, deploying the ORef model in Web Services environment offers the benefits of the ORef model ensuring that the Web Services are engineered upon a stricter object model.

This paper is structured as follows: in §2 we examine related research projects; in §3 we introduce the ORef model through its terminology and semantics; in §4 we highlight the model contribution; in §5 we examine the general semantics of query language, and finally offering the conclusions in §6.

## 2 RELATED RESEARCH

Our overview of the related research examines projects focused on projects that (1) provide generic frameworks for sensors management and querying and (2) projects that examined, designed and enhanced object models to store behaviour or to in-

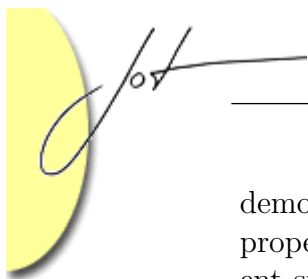


tegrate multiple data sources. The reason for choosing to examine (2) projects is that object models are more expressive than others as they permit defining behaviour in addition to the structure and thus are well suited for modelling complex environments such as sensors which are our target application area.

In the [RMS06] they provide a template for incorporating non-XML sources using XML thus tackling the same issues encountered by the sensor research area: converting the data into format that is usable for integration. They define a Data Format Description Language using which an XML representation of data is generated upon request. We feel that this approach is limited to read-only applications as the mapping does not permit defining an identification of the original data which thus cannot be referenced and updated; similarly, we do not see a possibility to define behaviour which is crucial for our target application area. The ORef model provides a unique identification of both objects and their properties that is similar to the various numbering schemes advised with storing XML nodes, primarily elements, attributes, text nodes and various other XML nodes as for example in [TVB<sup>+</sup>02].

In the COCOON project [SLR<sup>+</sup>94] a fully encapsulated object model was deployed in which an object's state is accessed using *functions*, further classified into *properties* and *methods*. A property may not modify an object's state and may be *stored* or *computed*. A computed property is either *derived*, when defined using a proprietary COOL programming language, or *foreign*, when defined using another, general-purpose programming language. A method updates the object's state. No further details on behaviour definition were published. The functions available with an object are specified in the definition of its *type*. A type *inherits* all functions from its *supertype* where additional functions may be defined. A *class* is a set of objects of a type where multiple classes of the same type may be defined and also, an object may take part in multiple classes. Additionally, for a type that has subtypes, *subclasses* that contain objects of corresponding subtypes may be defined, introducing apparently separate, but complexly linked hierarchies of types and classes which are hard to maintain. The query language operations provided in the model are rather simplistic and are restricted to a single class. The exception is the *extend* operation which permits defining new properties of the class that result from the query by accessing properties of a related object using a programmatic interface. Such capability is powerful, but however it is too complex to perform simple joins. Also, the full encapsulation introduces a performance penalty when compared to the direct access to properties available with the ORef model.

In the MultiView project [Run92], the SmallTalk object model was extended to provide multiple inheritance by storing an object's properties across multiple *implementation objects* using an *object-slicing* technique. The object-slicing remained transparent to client applications thanks to: (1) the full encapsulation of properties, present with SmallTalk, and (2) generated accessor operations for properties that for an accessed property identify the corresponding implementation object. The technique also forms the theoretical basis for providing objects that result from query evaluation as their properties may belong to different source objects. It was also



demonstrated that this technique improves the performance of client applications as properties that are not accessed are not read hence reducing the access to the persistent storage. The application of the technique beyond the SmallTalk programming language was not investigated. Due to these benefits it provides, the object-slicing technique is relevant to our work and is built into the ORef model. However, our approach applies and extends the original work in a more generic manner.

In the MOOD project [DOA<sup>+</sup>94], the C++ object model was used as the storage model. The behaviour was defined using the C++ programming language, compiled into dynamic libraries and loaded on request. SQL was used as the query language and extended to provide query access to objects. However, no behaviour could be invoked as part of the queries. The project provided a solid technical basis for object database kernels, however, no issues related to the object database model were addressed. Similarly, we use dynamic linking and loading of libraries as part of the implementation of the ORef model.

In the LOQIS project [SKL95], a mathematical object model was designed in which each object is assigned with a unique identifier. The state and behaviour of an object is determined by the object's *type* where an object of a simple type contains a value, an object of a collection type contains the identifiers of the objects that are in the collection, and finally, an object of a complex type contains the identifiers of the objects that take the role of the object's properties. Thus, both objects and their properties are given unique identifiers and are *addressable*. Such a feature is desirable with canonical models and correspondingly, is built into the ORef model also. However, the project did not examine an application of the LOQIS model as a general canonical model. Behaviour may be defined using the proprietary SBQL query language which is rather complex for simple operations. In their later work [KLS03] they examine updatable views that propagate updates to the original objects by means of stored procedures. We do not follow this approach as it requires defining many individual procedures that permit updating the original objects.

### 3 THE OREF MODEL

In this section, we introduce our Object-Reference model (ORef) which is an object-oriented model enhanced with *references* as the only feature capable of addressing objects and their properties. Further to this extension, the basic modelling primitives were respecified to include references. In this section, we clarify our terminology and we examine fundamental terms and definitions of the ORef model. These terms include object, types, behaviour, states, references, collections, user-defined types, relationships, inheritance and classes.

Our examples are based upon simplified sensor database schema taken from [Leg07] and illustrated in *figure 1*. In these experiments, readings from over 350 sensor devices were converted to XML and placed in a peer-to-peer environment (with output from one device on each peer). Queries against the entire sensor

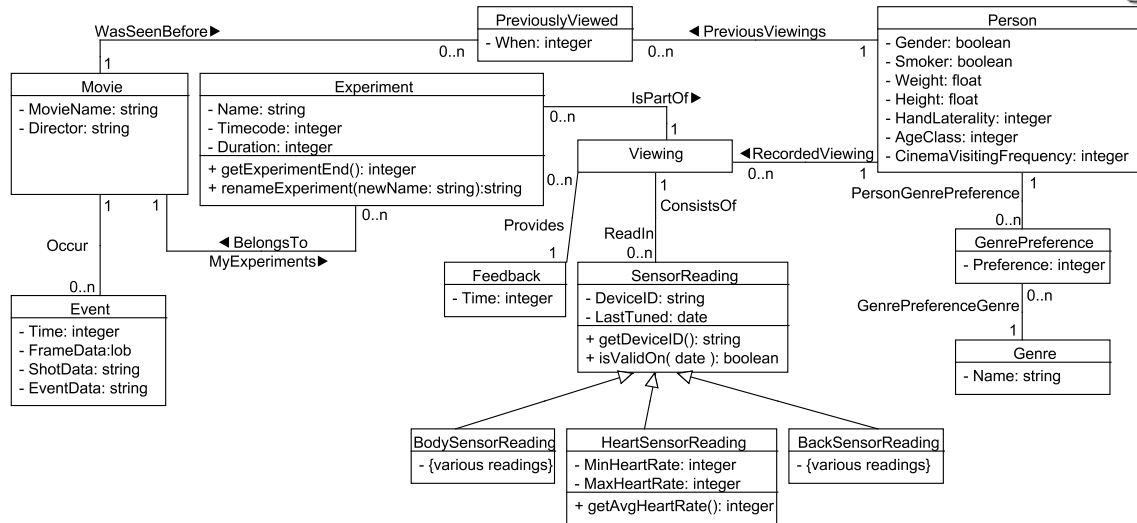


Figure 1: Sensor Schema Database UML Diagram.

network were slow and optimisation required the dynamic integration of smaller groups of devices for different query types. This dynamic integration proved difficult without a formal model to create the necessary mappings. Furthermore, normalisation of sensor data (for example, to synchronise timing across devices) required an update to the data generated by local sources, through the global schema. Later in this paper, we discuss how updates are facilitated using ORef.

## Basic definitions

An *object*  $o$  is an instance of a *type*  $\mathcal{T}$  which is denoted as  $o : \mathcal{T}$ . The type  $\mathcal{T}$  consists of the *structure*  $\mathcal{S}(\mathcal{T})$  and the *behaviour*  $\mathcal{B}(\mathcal{T})$ . An object's *state*  $state(o)$  must conform to the structure of the object's type, and likewise, an applied operation *behaviour*( $o$ ) must conform to the behaviour of the object's type. The state and the behaviour are the basic characteristics of any object. A unique *reference*  $ref(o)$  can be obtained to any *object*  $o$ , where this reference is the only addressing mechanism available in the model.  $\mathbb{T}$  is the set of all available types where a type is either *built-in*, *user-defined* or *collection*.

An object  $o$  of a built-in type has an *atomic* state from the modelling viewpoint, i.e. it is a single concept which cannot be divided any further. However, this does not prevent using an operation from extracting a portion of the state. For example, a month portion of a *date* object can be extracted though the state is still not dividable. We define *boolean*, *integer*, *float*, *date*, *character* and *lob* built-in types which are well-known, and we omit their full definitions.

In *example 1* two objects of simple types are materialised: an object  $o_1$  of the type *string* with the value 'HSR-10-G' and an object  $o_2$  of the type *date* with the value '2006-09-23'. For the sake of simplicity, the values of the references in this

paper will correspond to the object's subscript, i.e.  $ref(o_2) = 2$ .

$o_1:string:=\text{'HSR-10-G'}$

$o_2:date:=\text{'2006-09-23'}$

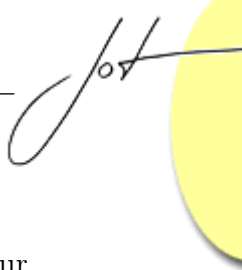
Example 1: Three objects materialised from built-in types.

An operation  $m$ , defined on type  $\mathcal{T}$ , transforms the state of an object  $o$  into  $o'$  where  $o$  and  $o'$  conform to the type  $\mathcal{T}$ . This transformation is denoted as  $transform(o, o')$ , which conforms to the behaviour of the type  $\mathcal{B}(\mathcal{T})$ . For example, the expression  $o_2.getMonth()$  invokes an operation  $getMonth$  against the  $o_2$  object, extracting the month portion of the date stored in the  $o_2$  object which remains unmodified. The transform expression for this operation is denoted as  $transform(o_2, o'_2)$ . Similarly, an expression  $o_2.advanceToNextDay()$  advances the date stored in the  $o_2$  object by one day resulting in  $o'_2$ . Correspondingly, the transformation expression is  $transform(o_2, o'_2)$ . In summary, the major benefit of the ORef model is that it is closed with regards to behaviour. This allows any object to have behaviour.

## Collections

A collection object  $O : \mathcal{T}_1$  is a container for (it *contains*) multiple objects  $o_i$  of type  $\mathcal{T}_1$ . The state of a collection is of a set of references to the objects that are contained in the collection i.e.  $state(O) = \{ref(o_1), ref(o_2), \dots, ref(o_n)\}$ . An expression  $base(\mathcal{T}_1)$ , defined only on collection types, determines the underlying type of the contained objects. A collection is either a *single*, a *list*, a *set* or a *bag* collection type. A single collection contains only one object. A list contains multiple objects and preserves the order of the objects in the collection. A set contains multiple objects where an object may not be contained in the same set more than once and the order of the contained objects is not preserved. Finally, a bag contains multiple objects some of which may be identical where their order is not preserved.

Collections have three basic capabilities: (1) to test whether an object is already *contained* in the collection, (2) to *include* an object and (3) to *exclude* an object. For example, for a collection  $O : \mathcal{T}_1$  that is a set of *Event* objects, i.e.  $\mathcal{T}_1 = set(Event)$ , the base type of the collection is  $base(\mathcal{T}_1) = Event$ . Assuming an  $o_1 : Event$  object that is not in the collection  $O$ , the expression  $O.contains(o_1)$  will result in a *boolean* object that evaluates to false; otherwise it will result in a *boolean* object that evaluates to true. The expression  $O.include(o_1)$  transforms a collection  $O$  that did not contain an object  $o_1$  into a collection  $O'$  that contains all objects contained in  $O$  and the object  $o_1$ . Similarly, the expression  $O.exclude(o_1)$  transforms a collection  $O$  that contains the object  $o_1$  into a collection  $O'$  that contains all original objects of  $O$  but not the object  $o_1$ . These three functions enable operating any object of type *set* in full.



## User-defined types

User-defined types are types that have *complex structure* and for which behaviour may be defined and thus the basic model definitions applicable to types apply also to user-defined types. The structure of a user-defined type (UDT) is a set of *properties* each of which has a unique name and a type. We denote the structure of a user-defined type as follows:  $\mathcal{S}(\mathcal{T}) \hat{=} \{(name_1, \mathcal{T}_1), \dots, (name_n, \mathcal{T}_n)\}$ . Consequently, the state of a *complex object* (i.e. an object of an UDT) is a set of named references, each pointing to the object that takes the role of the property with corresponding name, denoted as  $state(o) \hat{=} \{(name_1, ref(o_1)), \dots, (name_n, ref(o_n))\}$ .

For any complex object  $o$ , the object that takes the role of its property  $name$  may be obtained using  $o.name$ . These objects that take the role of properties are fully independent of their complex object. This characteristic of the ORef objects is similar to *object-slicing* introduced in [Run92] where their discussion was restricted to SmallTalk. This characteristic provides the basis of the query language semantics (QLS) discussed in §5.

In the *example 2* we define the structure of the *SensorReading* type that consists of properties *DeviceID* and *LastTuned* which are of *string* and *date* types respectively. For the sake of simplicity, in this example we ignore the *relationships* that the type has to type *Viewing* which is addressed in *Relationships* subsection, and also the fact that the type is a base type which is addressed in *Inheritance* subsection. An object  $o_4$  of the type *SensorReading*  $o_4$  uses objects  $o_1$  and  $o_2$  as properties.

$$\begin{aligned} S(\text{SensorReading}) &\hat{=} \{('DeviceID', \text{string}), ('LastTuned', \text{date})\} \\ o_1 : \text{string} &= 'SNSR-10' \\ o_2 : \text{date} &= '2006-09-27T18:50' \\ o_4 : \text{SensorReading} &= \{('DeviceID', ref(o_1)), ('LastTuned', ref(o_2))\} \end{aligned}$$

Example 2: The definition of type *SensorReading* and an object of this type.

In the *example 3*, the behaviour aspects of the type *SensorReading* are denoted by  $\mathcal{B}(\mathcal{T})$ , i.e. this is an operation signature for the type. We first declare the behaviour of the type *SensorReading* which consists of: (1) an operation *getDeviceID* that receives no parameters and returns a *string* object that contains the identification of the device, and (2) an operation *isValidOn* that receives a date and validates whether the sensor may be used for subsequent reading at the specified date or it must be tuned before. Then we demonstrate the invocation of the operation *getDeviceID*, define a *date* object, and finally, invoke *isValidOn* ignoring the return value.

## Relationships

A binary relationship  $\mathcal{R}$  is an association between two user-defined types  $\mathcal{T}_A$  and  $\mathcal{T}_B$  modelled as two collection properties: (1) *property* $_{A \rightarrow B}^{\mathcal{R}}$  that belongs to  $\mathcal{T}_A$ , and

$$B(\text{SensorReading}) \hat{=} \{(\text{getDeviceID}, (), \text{string}), (\text{isValidOn}, (\text{date}), \text{boolean})\}$$

$$o_4.\text{getDeviceID}()$$

$$o_5 : \text{date} = \text{'2007-09-27T18:50'}$$

$$o_4.\text{isValidOn}(\text{ref}(o_5))$$

Example 3: Behaviour of the type *SensorReading*.

(2)  $\text{property}_{B \rightarrow A}^{\mathcal{R}}$  that belongs to  $\mathcal{T}_B$ . For each object  $o_A$  of type  $\mathcal{T}_A$ , the property  $\text{property}_{A \rightarrow B}^{\mathcal{R}}$  contains references to all *related* objects  $o_B$  of the type  $\mathcal{T}_B$ . Similarly, the same applies to each object  $o_B$  of  $\mathcal{T}_B$ , its respective  $\text{property}_{B \rightarrow A}^{\mathcal{R}}$  and corresponding  $o_A$  of  $\mathcal{T}_A$ . For each relationship  $\mathcal{R}$ , *referential integrity* is preserved which means that an object  $o_A$  is related to  $o_B$  via  $\text{property}_{A \rightarrow B}^{\mathcal{R}}$  if and only if the object  $o_B$  is related to  $o_A$  via property  $\text{property}_{B \rightarrow A}^{\mathcal{R}}$ . Maintaining referential integrity is straightforward due to the mechanism of references built into the model as explained in §4.

In *example 4* as we are only interested in the relationship properties, we do not elucidate the behavioural properties denoted by  $\mathcal{B}(\mathcal{T})$ . We provide definitions of types *Viewing* and *SensorReading*, and a relationship *SensorReadingsInViewing* that specifies all sensor readings for a single viewing of a movie clip. From this example, for the sake of simplicity, we omit all other relationships class *Viewing* has with other classes. Furthermore, the example also does not consider that *SensorReading* is a *base class* which is addressed in *Inheritance* subsection. In this example, a *Viewing* object  $o_7$  is related to a *SensorReading* object  $o_{11}$ . This relationship, from *Viewing* to *SensorReading* is named *ConsistsOf* and its traversal, from *SensorReading* to *Viewing* is named *ReadIn*.

$$\text{Viewing} \hat{=} (S(\text{Viewing}), B(\text{Viewing}))$$

$$\text{SensorReading} \hat{=} (S(\text{SensorReading}), B(\text{SensorReading}))$$

$$\text{SensorReadingsInViewing} \hat{=} \mathcal{R}(\text{'ConsistsOf'}, \text{set}(\text{SensorReading}),$$

$$\quad \text{'ReadIn'}, \text{single}(\text{Viewing}))$$

$$S(\text{Viewing}) \hat{=} \{\text{SensorsReadingsInViewing}(\text{Viewing})\}$$

$$S(\text{SensorReading}) \hat{=} \{(\text{'DeviceID'}, \text{string}), (\text{'LastTuned'}, \text{date}),$$

$$\quad \text{SensorReadingsInViewing}(\text{SensorViewing})\}$$

$$o_6 : \text{SensorsReadingsInViewing}(\text{Viewing}).\text{type} = \{\text{ref}(o_{11})\}$$

$$o_7 : \text{Viewing} = \{(\text{'ConsistsOf'}, \text{ref}(o_6))\}$$

$$o_8 : \text{string} = \text{'SNSR-3'}$$

$$o_9 : \text{date} = \text{'2007-10-01'}$$

$$o_{10} : \text{SensorsReadingsInViewing}(\text{SensorReading}).\text{type} = \{\text{ref}(o_7)\}$$

$$o_{11} : \text{SensorReading} = \{(\text{'DeviceID'}, \text{ref}(o_8)), (\text{'LastTuned'}, \text{ref}(o_9)),$$

$$\quad (\text{'ReadIn'}, \text{ref}(o_{10}))\}$$

Example 4: The relationship *SensorReadingsInViewing*.





## Inheritance

The model supports *single inheritance* which is a feature of the model whereby a new type  $\mathcal{T}_{sub}$  can be defined to *inherit* from an immediate original type the following: (1) the structure, in terms of all original properties inclusive of original relationships, and (2) the behaviour.  $\mathcal{T}_{sub}$  may optionally extend the original type with new properties and behaviour and thus, is a *subtype* of the original type  $\mathcal{T}$ . Any object  $o_{sub}$  of the subtype  $\mathcal{T}_{sub}$  may substitute for an object  $o$  of the original type  $\mathcal{T}$ , as it features all properties and operations supported by objects of the type  $\mathcal{T}$ . The inheritance mechanism may not be used to remove the original properties or operations, nor to modify the names or types of the original properties. However, operations may be redefined thus facilitating *polymorphism*.

In *example 5*, we first repeat the definitions of types *Viewing* and *SensorReading*, and also the definition of the relationship *SensorReadingsInViewing* from *example 4*. Then, we define the type *HeartSensorReading* that inherits from *SensorReading* hence it contains all the properties and has all behaviour of the type *SensorReading* namely properties *DeviceID* and *LastTuned*, and the relationship to type *Viewing*. Additionally, the type *HeartSensorReading* also has properties *MinHeartRate* and *MaxHeartRate* that contain the heart rate reading of the sensor. Finally, when an object of the type *HeartSensorReading* is accessed as an object of type *SensorReading*, only *DeviceID*, *LastTuned* and *ReadIn* are visible.

## Classes

An *extent* of a type  $\mathcal{T}$  denoted as  $ext(\mathcal{T})$  is a set that contains objects that are only of the type  $\mathcal{T}$  and of no other type thus excluding the objects of subtypes of  $\mathcal{T}$ . Such an extent is also referred to as *shallow*, contrary to a deep extent  $deepExt(\mathcal{T})$  that contains all objects that are of the type  $\mathcal{T}$  including objects of subtypes of  $\mathcal{T}$ . A *class* of a type  $\mathcal{T}$  is an ordered pair consisting of a type  $\mathcal{T}$  and the extent that corresponds to the type  $\mathcal{T}$ .

## 4 MODEL CONTRIBUTION

In this section, we highlight the contribution of the ORef model by describing the model-specific enhancements. Our discussion covers: (1) direct addressability of objects and properties, (2) direct access to properties, (3) maintaining referential integrity of relationships, (4) ORef operations, (5) modelling aggregations, (6) replacing multiple inheritance, and finally (7) subclassing.

$$\begin{aligned}
\text{Viewing} &\hat{=} (S(\text{Viewing}), B(\text{Viewings})) \\
\text{SensorReading} &\hat{=} (S(\text{SensorReading}), B(\text{SensorReading})) \\
\text{SensorReadingsInViewing} &\hat{=} \mathcal{R}('ConsistsOf', \text{set}(\text{SensorReading}), \\
&\quad 'ReadIn', \text{single}(\text{Viewing})) \\
S(\text{Viewing}) &\hat{=} \{\text{SensorsReadingsInViewing}(\text{Viewing})\} \\
S(\text{SensorReading}) &\hat{=} \{('DeviceID', \text{string}), ('LastTuned', \text{date}), \\
&\quad \text{SensorReadingsInViewing}(\text{SensorReading})\} \\
\text{HeartSensorReading} &\hat{=} (S(\text{HeartSensorReading}), B(\text{HeartSensorReading})) \\
&\quad \triangleright \text{SensorReading} \\
S(\text{HeartSensorReading}) &\hat{=} S(\text{SensorReading}) \cup \\
&\quad \{('MinHeartRate', \text{integer}), ('MaxHeartRate', \text{integer})\} \\
o_{12} : \text{string} &= \text{'HR-SNSR-12'} \\
o_{13} : \text{date} &= \text{'2007-10-01'} \\
o_{14} : \text{SensorsReadingsInViewing}(\text{SensorReading}).\text{type} &= \{\text{ref}(o_{17})\} \\
o_{15} : \text{integer} &= 122 \\
o_{16} : \text{integer} &= 150 \\
o_{17} : \text{Viewing} &= \{('ConsistsOf', \text{ref}(o_{18}))\} \\
o_{18} : \text{SensorsReadingsInViewing}(\text{Viewing}).\text{type} &= \{\text{ref}(o_{19})\} \\
o_{19} : \text{HeartSensorReading} &= \{('DeviceID', \text{ref}(o_{12})), ('LastTuned', \text{ref}(o_{13})), \\
&\quad ('ReadIn', \text{ref}(o_{14})), \\
&\quad ('MinHeartRate', \text{ref}(o_{15})), ('MaxHeartRate', \text{ref}(o_{16}))\} \\
\text{Note} : \text{When viewed as SensorReading object, } o_{19} &\text{ does not have properties} \\
&\quad \text{MinHeartRate and MaxHeartRate.} \\
o_{19} : \text{HeartSensorReading} &= \{('DeviceID', \text{ref}(o_{12})), ('LastTuned', \text{ref}(o_{13})), \\
&\quad ('ReadIn', \text{ref}(o_{14}))\}
\end{aligned}$$

Example 5: *HeartSensorReading* type that inherits from *SensorReading*.

## Direct Addressability of Objects and Properties

Objects and properties in the ORef model are addressable using the single paradigm of references. References use the unique identification of both objects and properties provided using an *orefOID*. Namely, each object  $o$  is assigned with an *orefOID* which is an intrinsic feature of each object and provides a unique identification of the object with no other associated or implied meaning. Thus, two objects are *identical* if and only if they both have the same *orefOID*. Throughout this paper, we adopt a convention that an *orefOID* of an object  $o_i$  is  $i$  and also that each  $i$  is a plain integer. A reference  $\text{ref}(o_i)$  points to an object  $o_i$  by embedding its *orefOID*, or simply storing  $i$ . While references conceptually model the notion of pointing to objects from a user's point of view, *orefOIDs* are the internal model mechanism that achieves the physical of effect. In [KRM07] we demonstrated that *orefOIDs* play the central role in transparent transformation of database objects into ORef objects, and also provide access to remote ORef objects that are not available on the same ORef server.



## Direct Access to Properties

An object used as a property of any complex object may be obtained by specifying the complex object and the name of the required property. The expression  $o_{19}.DeviceID$  in *example 5* results in obtaining a reference to object  $o_{12}$  (*DeviceID*) that takes the role of the property *DeviceID* of object  $o_{19}$ .

## Maintaining Referential Integrity of Relationships

The direct addressability of properties introduced in previous subsection forms the basis for maintaining the referential integrity of relationships. Assuming a relationship *SensorReadingsInViewing* from *example 4*, any modification to *ConsistsOf* property of  $o_7$  object must also result in updating the property *ReadIn* of all related (*SensorReading*) objects, which in our case is only  $o_{11}$ . This is a straightforward process as the *ConsistsOf* property of each *Viewing* object contains all related *SensorReading* objects. Due to the direct addressability of properties, it is possible to obtain the traversal property for each such related *SensorReading* objects and update accordingly.

## Operations

References interface ORef operations and objects as: (1) an operation receives both the target object and parameter objects as references, and (2) evaluates a single object that is returned as a reference. Thus, ORef operations are closed in the ORef canonical model. Implementation of references includes transparent reading of objects and their materialisation on-demand, allowing ORef operations to be deployed against persistent objects.

## Modelling Aggregations

A property may only be of a built-in or a collection type, but not a user-defined type. Some other object-oriented models support properties of user-defined types to model *aggregations* where the object aggregated to the parent object is deleted once the parent object is removed. To model an aggregation in the ORef model, (1) an aggregated object is connected to its parent object using a relationship, and (2) the behaviour that deletes the parent object is defined also to delete all aggregated objects.

## Replacing Multiple Inheritance

*Multiple inheritance* is a feature present in some object-oriented models whereby a type is allowed to inherit from more than one type. This feature introduces unne-

cessary modelling complexities such as the *dreaded diamond* which occurs when a type  $A$  is inherited by two different subtypes  $B$  and  $C$  which again are inherited by type  $D$  resulting in property propagation conflicts. From a modelling perspective, multiple inheritance can be easily avoided by using other modelling primitives hence it was intentionally omitted from many object-oriented models, and also from our work in the ORef model.

## Subclassing

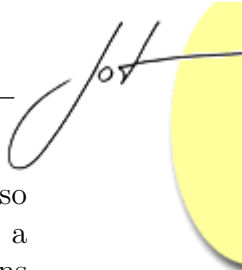
We do not introduce *subclassing* which allows the definition of a subclass of an existing class, but which is really only a subset of the extent of the type, e.g. SQL:1999 [GP99]. This separation between classes and types introduces unnecessarily complex interconnections between type and class hierarchies. As similar effect may be achieved using views, this is the effect achieved in the ORef model.

## 5 QUERY LANGUAGE SEMANTICS

In the previous sections, we illustrated the benefits introduced by the ORef model. We now examine the *query language semantics* (QLS) which is a set of rules that any query language must follow to be compliant with the ORef model. *EQL* [KBR03], an *EGTV Query Language* used in the EGTV research implements these rules. In the remainder of this chapter, we introduce the semantics and examine its relationship with the model providing examples using EQL.

The query language semantics is based upon *object-generating semantics* (OGS) that requires a set of *objects* with new object identifiers to be generated to represent the query result. Kim et al [KK95] examine object-generating semantics in relation to *object-preserving semantics* (OPS) and demonstrate that deploying OPS is crucial for ensuring query result updatability. However, OPS is not compatible with current object-oriented models as generated objects preserve the identity of their sources. This further implies that both the generated and the original object have the same identity, yet may be of a different type which violates the principle of object identifier uniqueness, hence our decision to choose an OGS. In our semantics, we deploy OPS using properties in a manner that preserves object identifier uniqueness as explained in §5 which is facilitated by the ORef model. Hence we preserve compatibility with current object-oriented models, but we also enable update propagation.

The QLS specifies that a query results in a single class that, as per our ORef model class definition in §3, is an ordered pair of a type and an extent. All generated objects in the query result conform to the result type and they belong to the result extent. As described in Classes subsection of §3 this extent is a set object that is returned to the client application using a reference. This implies that no special interface is deployed to process classes and objects that are query results. The direct benefit is that client applications are not aware whether they process objects stored



by the database, or objects that result from queries (closure property). This also provides a solid theoretical basis for nesting queries as the result of any query is a class, and thus, it can be nested in another query requiring no special considerations as is the case with other models where the query might result in a set of structures, literals or objects for example, the ODMG [CB99].

## Specifying and evaluating a query

Following the traditional approach to query languages, we specify that a query consists of *source*, *projection* and *restriction* clauses where: (1) *source* clause lists classes that participate in a query and determines the set of valid expressions that can be used in *projection* and *restriction* clauses; (2) *projection* clause establishes the structure  $S(\mathcal{T})$  of the resulting type  $\mathcal{T}$  in terms of its properties where each property is a named expression; and finally (3) all resulting objects  $o$  must comply with the condition given in the *restriction* clause which is a boolean expression. The resulting type  $\mathcal{T}$  has no behaviour as the original behaviour was defined only for objects of the original type, and thus cannot be applied against the generated objects as they may have different structure.

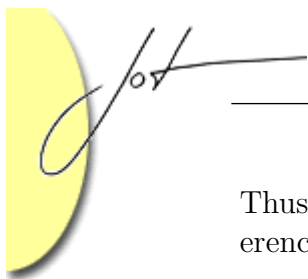
To materialise resulting objects  $o$ , the *restriction* clause is evaluated for each element in the Cartesian product of *source* classes, and if this clause evaluates to true, an object  $o$  is materialised with the structure specified in the *projection* clause. We now describe the clauses in detail and then we examine the deeper implications of such a query definition.

**Definition 5.1** *A query  $Q(\text{projection}, \text{source}, \text{restriction})$  transforms classes listed in a source clause into a new class such that the structure of its type  $S(\mathcal{T})$  is given by the projection clause. The type has no behaviour  $B(\mathcal{T}) = \emptyset$ . And the restriction clause specifies the condition that all generated objects must satisfy.*

### Source clause.

The classes that participate in a query are listed in the source clause along with their corresponding *aliases*. A new alias permits a class that is previously listed in the source clause of a query to participate as a different role. If omitted, an alias defaults to the class name. Aliases are used to form expressions in *projection* and *selection* clauses, thus each alias evaluates to one of the objects that belongs to the containing class. We shall further explain this concept using *examples 6 and 7*.

In *example 6* class *HeartSensorReading* under alias *hsr* is listed as the only class in the source clause, and projection and restriction clauses are evaluated for each object of this class. Throughout the query evaluation, the alias *hsr* will contain a reference to an object of class *HeartSensorReading*. Similarly in *example 7*, classes *HeartSensorReading* and *Viewing* are listed as *hsr* and *vw* respectively, and evaluation is performed for each element of the Cartesian product of these two classes.



Thus, throughout the query evaluation, aliases *hsr* and *vw* will each contain a reference to an object of classes *HeartSensorReading* and *Viewing* respectively.

**FROM** HeartSensorReading hsr

Example 6: A source clause containing *HeartSensorReading* class.

**FROM** HeartSensorReading hsr,  
Viewing vw

Example 7: A source clause containing *HeartSensorReading* and *Viewing* classes.

### Projection clause.

The projection clause *projection* contains a list of *named expressions*  $e_j$  that determine the structure  $S(T)$  of the resulting type  $T$ . When expression names are omitted, implementation query languages such as EQL provide a default naming convention. For the purposes of our discussion, we shall assume we can always name the expressions and that these names are available.

When a query is evaluated, a query result object is materialised which contains the generated objects. Each generated object contains the projected properties.

There are two basic restrictions with  $e$  expressions. Firstly, it is not possible to change the type of an expression. Secondly, it is not possible to perform projections on relationships as this would violate the model restriction that each relationship is bi-directionally navigable.

These limitations can be overcome using behaviour expressions to *simulate* changing the resulting type of a property, i.e. an operation is invoked which receives the original property, and this operation transforms the original property type into the desired type. For example, assuming that a *date* is transformed to a *string*, a user-defined type *DateString* that inherits from a *string* is defined, and an operation that performs the translation is provided. This operation receives a *date*, and returns a *DateString* object materialised to represent the date as a *string*. A similar approach is used to transform a bi-directional relationship property into a uni-directional property which is then aggregated by the resulting object to overcome the second limitation we noted previously. Essentially, this establishes a uni-directional relationship between the resulting query class and source class instead of a bi-directional relationship which would violate the model semantics.

### Restriction clause.

The restriction clause *restriction* is an expression that generates a reference to a *boolean* object. This expression is evaluated before materialising the result object, and if it evaluates to `true`, the corresponding result object is materialised and added to result extent, otherwise it is just discarded. The restriction clause may be omitted and then all result objects are added to query extent. Multiple expressions can be connected using logical operators defined for a *boolean* type thus evaluating



a single *boolean* object. This alleviates the need to define logical operations as part of the QLS, and relies instead on the corresponding model definition. In *example 8* we process only the heart sensor readings that average below 150.

**WHERE** `hsr.getAvgRate() < 150`

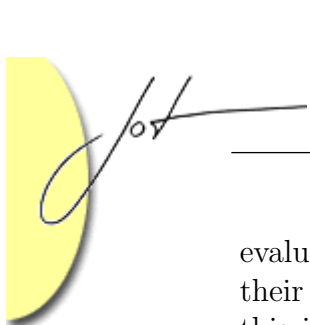
Example 8: Filtering for all experiments that are executed in the beginning of a movie.

## Benefits of the Query Language Semantics

Query languages that follow our QLS may provide object manipulation using only the object's properties and operations. Specifically, a query language may not introduce a new operation to transform objects. Instead, this effect is achieved by extending the corresponding type with additional operations that are then invoked from the query language. This restriction prevents a query language from introducing an operation that is not compliant with the model definition. Additionally, this restriction promotes code-reuse as any new operation must: (1) be developed for the stored type at the correct model level, thus is accessible to all applications directly; (2) invoked through the query language processor, the same any other operation. From a practical viewpoint and purely as an optimisation technique, a query processor may re-implement some operations to improve performance. Typically, boolean operations would be re-implemented and improved using well known *heuristics*, for example, not evaluating the right side of boolean operator (*and*), if the left side has already evaluated to *false*.

In the introduction of §5 we argued our preferences towards OGS as the basis for our QLS, as OGS requires objects with new identifiers to be generated from each query, which then preserves the principle of object identifier uniqueness. This is the opposite to OPS, which requires the resulting objects to preserve the identity of objects used in their materialisation (introducing the notion that multiple objects share the same identity). The difficulty that arises here, however, is that, though an object identifier still identifies a single object, it does not identify its own type as the object belongs to multiple types and classes. For example, assuming classes *Person* and *Adult*, whereby adults are persons over a certain age, then an adult person is represented by an object that belongs to both classes, and is accessible through both interfaces, where both person's and adult's type interface are applicable. Determining which interface needs to be applied to the particular object depends purely on the context which then needs to be provided along with the object identification. Current object models do not support such a configuration as they presume a single interface per object and per type. Additionally, we see that such a combined object identification itself is a new complex identity, and hence internally implements OGS which supports our decision to implement OGS.

We deploy an OPS with regards to properties as these objects result from the



evaluation of expressions and are aggregated into resulting objects without changing their type as demonstrated in query structure subsection of §5. The implication of this is that no additional type identification is required thus avoiding unnecessary complexity.

This combination of OGS in relation to resulting objects, and OPS in relation to properties of resulting objects, permits us to preserve compatibility with current object technologies while providing a solid basis for query result updatability. In essence, properties are re-aggregated by result objects, and thus any update to the resulting object is actually an update against the object used to materialise it. This means no additional processing is required. We now examine updatability issue in more detail.

## Updatability

The QLS defines that materialisation of the query result deploys OPS in relation to properties, which in turn provides updatability to result objects. Namely, properties result from the evaluation of expressions, hence updates applied against properties are applied to the source objects directly. The definition of this model relationship was a pre-requisite to clear specification of updates semantics. The only exception to full update capability occurs when the property results from an operation invocation. Then relationship between the target object the operation is invoked against, and result object that the operation returns, purely depends on the definition of the operation. Hence, to enable the update propagation, the object returned by an operation must maintain a link to the instance object to facilitate update propagation. This capability is supported by the ORef model as it allows for the original reference to the original object to which the query is targeted to be stored and furthermore it provides a capability to define new operations on the generated object.

### Modifying objects using queries

The query language semantics facilitates object modifications to be achieved using defined type operations. In the remainder of this section, we examine how traditional DML<sup>1</sup> operations namely INSERT, UPDATE and DELETE are achieved.

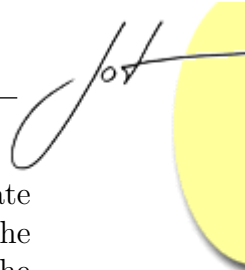
Our discussion first examines these three basic cases, and then we demonstrate that when practical, they can be combined in the same query. This results in a reduction of the amount of code written and its complexity.

**Creating objects.** The creation of objects corresponds to INSERT DML operation, and is achieved by invoking a constructor operation as a part of the projection clause. For each result object, and corresponding contained object within the result

---

<sup>1</sup>DML stands for Data Manipulation Language which is a commonly found section of the query language that permits modifying stored data using the query language.





object, references are generated by the constructor. In *example 9* we demonstrate the creation of a single object of type *HeartRateReading* using a constructor. The corresponding query has a type that corresponds to the projection clause of the query. The result object extent contains a single object namely the object that contains the newly created *HeartRateReading* object. In *example 10* we demonstrate the semantics of `INSERT...AS SELECT` by creating a corrected copy of the original *HeartRateReading* objects for sensor HR-SNRS-12. Similarly to *example 9*, the resulting extent contains a generated object for each created *HeartRateReading* object; this generated object has a single collection property that contains the corresponding created *HeartRateReading* object.

```
SELECT HeartSensorReading('HR-SNRS-12','2007-10-01', 80, 124)
FROM   HeartSensorReading
```

Example 9: Create new heart rate sensor reading.

```
SELECT HeartSensorReading.HeartSensorReading(
    hsr.DeviceID + ' Corrected', hsr.LastTuned,
    hsr.MinHeartRate - 5, hsr.MaxHeartRate - 5)
FROM   HeartSensorReading hsr
WHERE  hsr.DeviceID='HR-SNRS-12'
```

Example 10: Create corrected copies of the original heart sensor readings.

**Updating objects.** The QLS equivalent of the DML operation `UPDATE...SET`, provides the same effect by invoking modification operations on properties as part of the projection clause, i.e. the result object contains references to the modified properties. In *example 11*, all *HeartSensorReading* objects with the *DeviceID* of 'HR-SNRS-12' is updated: their *DeviceID* is extended to include word 'Corrected', and minimum and maximum heart rate decreased by 5. The result set contains a single generated object for each modified *HeartSensorReading* object; this generated object has a three properties: *DeviceID*, *MinHeartRate* and *MaxHeartRate* each containing the corresponding modified property.

```
SELECT hsr.DeviceID := hsr.DeviceID + ' Corrected',
    hsr.MinHeartRate := hsr.MinHeartRate - 5,
    hsr.MaxHeartRate := hsr.MaxHeartRate - 5
FROM   HeartSensorReading hsr
WHERE  hsr.DeviceID='HR-SNRS-12'
```

Example 11: Correct *HeartSensorReading* sensor readings.

**Deleting objects.** Deleting an object corresponds to the DML `DELETE` operation, and it is achieved by invoking a destructor operation within the projection

```

SELECT hsr := nil
FROM HeartSensorReading hsr
WHERE hsr.DeviceID='HR-SNRS-12'

```

Example 12: Delete *HR-SNRS-12* sensor readings.

clause. The EQL syntax accomplishes this by assigning a `nil` to the alias. In *example 12* this operation is used to remove the *HeartSensorReading* with the *DeviceID* of 'HR-SNRS-12'. The resulting type has a single property named *hsr* which contains a reference to the deleted object. Should the operation be successful, this property will point to `nil` allowing to process exceptions in cases where a deletion of an object fails. This is outside of the scope of the current paper.

**Combining DML operations.** We have demonstrated that the query language semantics covers commonly found DML operations. However, the actual benefit from the semantics arises when DML operations are combined in a single query to achieve the effect of multiple queries. In *example 13*, for each *HeartSensorReading* object the *DeviceID* of which is 'HR-SNRS-12', its corrected copy is created and at the same time, the *DeviceID* of the original object is modified to include the word 'Original'. For each copied *HeartSensorReading* object, the generated object that belongs to the query result contains: (1) the copied *HeartSensorReading* object, (2) property *DeviceID* of the original *HeartSensorReading* object.

```

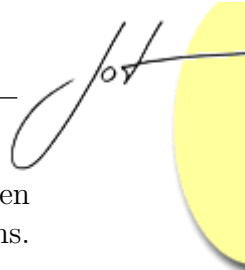
SELECT HeartSensorReading.HeartSensorReading(
    hsr.DeviceID + ' Corrected', hsr.LastTuned,
    hsr.MinHeartRate - 5, hsr.MaxHeartRate - 5),
    hsr.DeviceID + ' Original'
FROM HeartSensorReading hsr
WHERE hsr.DeviceID='HR-SNRS-12'

```

Example 13: Create a corrected copy of *HeartSensorReading* with the *DeviceID* of 'HR-SNRS-12' and update the *DeviceID* of the original one.

## Defining query class operations

The QLS requires that output from a query is a result object which has a class extent and in query structure subsection of §5 we described how resulting objects are materialised. This implies that the operations applicable to source objects cannot be used on resulting objects, as these objects are of different types. From user's viewpoint, objects resulting from a query are no different to other object, and no theoretical obstacles exist which would prevent further definition of operations, i.e. section of the model semantics relevant for operations elaborated in §4 applies to operations of result type. Thus, result type operations are defined using properties,



their operations, and the operations defined for the result type. However, it is often necessary to use the operations of source types to define the result type operations.

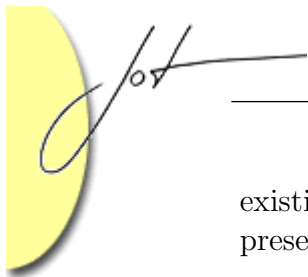
For example, returning to the *HeartSensorReading* type that inherits *getDeviceID* and *isValidOn* operations from *SensorReading* type defined in *example 3* and has own *getAvgHeartRate* operation, a *HighAvgHeartRateReadings* query that returns only the heart rate readings that average above 150 for which the *WHERE* clause is illustrated in *example 8* contains no operations. However, in order to verify whether these readings are caused by sensors that were not tuned properly at the time the reading was taken, an access to operation *isValidOn* must be provided. To facilitate this, each result object contains properties which take their names from the *aliases* that containing references to corresponding source objects, through which the operations of source types can be accessed. To provide a result object with access a source operation, a *wrapper* operation can be defined which extracts the corresponding source object, and invokes the source operation against it, where the wrapper operation is named after the source operation. For example, *isValidOn* operation would often be used for the *HighAvgHeartRateReadings* type.

## 6 CONCLUSIONS

This paper introduced the ORef model which is used as canonical model for integrating information systems. The ORef model is a standard object-oriented model enhanced with the paradigm of references in order to define closed transformations. This simply means that each transformation operates on one or more objects, and results in one or more objects, where transformations are operations and queries.

A reference points to an object or to its property, and is the only model construct capable of pointing to objects or their properties. Where appropriate, basic object-oriented modelling primitives are re-specified to include references, for example an object does not directly *contain* its properties, rather they are independent objects and are connected using references. References permit the pass of objects into an operation as the objects are unknown when the operation is defined except for their type. The references form the basis for all further communication between an operation and objects. The object that results from the operation is returned using its reference. This correlation between references and operations provides database independent definition of behaviour when the ORef model is needed to create the same operation on different databases or peers.

References also influenced the query language semantics that specified the rules for materialising the query result. The QLS is based upon the object-generating semantics which specifies that new objects result from a query and obtain new identifiers, thus preserving the uniqueness of object identifiers. Unique object identifiers permit the definition of operations for objects that result from queries, and in a similar manner, the definition of operations of stored objects, and also permits access to operations of source objects. Moreover, maximum compatibility with

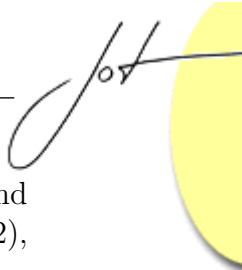


existing object-oriented systems is preserved. With regard to properties, the object-preserving semantics is deployed to enable updatability of the result objects.

While this paper focuses entirely on the object model and its properties, our current work focuses on the functionality of the model, and plans for future research include benchmarking an ORef implementation against its equivalent native object-oriented database application. This step includes the actual sensor network used to provide examples in this paper to develop a set of ORef virtual schemas and demonstrate both usability and performance in a peer-to-peer scenario.

## References

- [CB99] Catell, R. and Barry, D., *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann Publishers, 1999.
- [DOA<sup>+</sup>94] Dogac, A., Ozkan, C., Arpinar, B. et al., METU Object-Oriented DBMS, in *Object-Oriented Database Systems*, pp. 513–541, Springer-Verlag, 1994.
- [GP99] Gulutzan, P. and Pelzer, T., *SQL-99 Complete, Really*, R&D Books, 1999.
- [KBR03] Kambur, D., Bećarević, D. and Roantree, M., An Object Model Interface for Supporting Method Storage, in *Proceedings of the 7th East European Conference on Advances in Databases and Information Systems (AD-BIS)*, 2003.
- [KK95] Kim, W. and Kelley, W., On View Support in Object-Oriented Databases Systems, in *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pp. 108–129, ACM Press and Addison-Wesley, 1995.
- [KLS03] Kozankiewicz, H., Leszczyłowski, J. and Subieta, K., Implementing Mediators through Virtual Updateable Views, in *Proceedings of the 5th Workshop EFIS 2003*, pp. 52–62, IOS Press, 2003.
- [KRM07] Kambur, D., Roantree, M. and Murphy, J., Using an Object Reference Approach to Distributed Updates, in *18th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 182–191, Springer-Verlag, 2007.
- [Leg07] Legeay, N., Experimental Architecture and Sensor Data Descriptions, *Technical Report ISG-07-02*, Dublin City University, 2007, URL <http://www.computing.dcu.ie/~isg/publications/ISG-07-02.pdf>.
- [LRJ<sup>+</sup>07] Legeay, N., Roantree, M., Jones, G. J. et al., Semi-Automatic Enrichment of Raw Sensor Data, in *to appear in ODBASE 2007*, 2007.



- [RMS06] Rose, K. H., Malaika, S. and Schloss, R. J., Virtual XML: a toolbox and use cases for the XML world view, in *IBM Systems Journal*, vol. 45(2), pp. 411–424, 2006.
- [Run92] Rundensteiner, E. A., MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases, in *Proceedings of the 18th International Conference on Very Large DataBases (VLDB'92)*, Vancouver, British Columbia, Canada, pp. 187–198, Morgan Kaufmann Publishers, 1992.
- [SKL95] Subieta, K., Kambayashi, Y. and Leszczyłowski, J., Procedures in Object-Oriented Query Languages, in *Proceedings of the 21st International Conference on Very Large DataBases*, pp. 182–193, Morgan Kaufmann Publishers, 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.
- [SLR<sup>+</sup>94] Scholl, M., Laasch, C., Rich, C. et al., The COCOON Object Model, *Technical Report 211*, Dept of Computer Science, ETH Zurich, 1994.
- [TVB<sup>+</sup>02] Tatarinov, I., Viglas, S. D., Beyer, K. et al., Storing and querying ordered XML using a relational database system, in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 204–215, ACM, 2002.

## ABOUT THE AUTHORS

**Dalen Kambur** is a PhD student at the Dublin City University, Ireland and may be reached at [dalen.kambur@computing.dcu.ie](mailto:dalen.kambur@computing.dcu.ie). See also <http://www.computing.dcu.ie/~dalenk>.