Framework supporting Rapid Information Modelling

C. Ignat, M. C. Norrie

Institute for Information Systems
Swiss Federal Institute of Technology (ETH)
ETH-Zentrum, CH-8092 Zurich, Switzerland

Abstract

In this paper, we present a framework capable of supporting Rapid Information Modelling. The management of information is done at the conceptual level without the user having to define a data model for information organisation. We introduce the notion of heterogeneous collections together with a flexible notion of typing. One can easily define new typing constraints according to user needs and integrate them into the framework. To support the processing and querying of information, we provide algebra operations which can be evaluated with or without type checking. An initial version of the framework has been implemented as a web application offering flexible access to information.

1 Introduction

Information modelling can be understood as the activity of designing specific domains of interest called *information spaces* [Kob00]. Information spaces can be seen as environments into which users enter to find answers to questions about a specific topic, to browse large collections of information and to update and reorganise information. Information spaces may be shared by user communities and individual users may want to organise or view the information spaces according to their own preferences or requirements.

Rapid Information Modelling (RIM) supports the construction of information spaces on the conceptual level without it being necessary to define a data model for information organisation. In an RIM application, the user can represent concepts of a specific domain as information objects, create categories for these information objects, build relationships between them or add properties to them. The user can perform all these activities without knowing about information or data modelling. Information objects can be dynamically classified into heterogeneous collections of objects and associations. Further, the operations on these collections may optionally be performed with or without type restrictions.

For implementing our RIM framework, we have used the eXtreme Design (XD) framework developed by Kobler [Kob00] and the generic object model OM as an expressive data model for supporting the conceptual design modelling process.

Since web browsers are being used increasingly, not only for browsing the world-wide web, but also as a generic user interface for all kinds of web-applications, we decided to implement an initial prototype as a web application.

The RIM framework and the Web application implemented as an initial prototype of this framework represent the initial step in the research problem of rapid information modelling. The second stage is to develop the desktop application which will be described later in this paper. Based on the feedback from the second stage, we will evaluate our approach and carry out any necessary adaptations or refinements. In a third stage, we then plan to extend the framework to support general application development.

In this paper, we present the metamodel which underlies the RIM approach, together with the algebra developed for RIM. Additionally, we outline the architecture of our prototype and discuss the directions of future work in terms of both extensions to the framework and applications.

To motivate the RIM framework, we begin in section 2 with a brief statement of our vision expressed in terms of a planned first application. Section 3 presents an overview of the OM model and XD design, emphasising the reasons for choosing them in our approach. Section 4 motivates the need for information modelling by means of an example and presents the overall architecture describing classification and typing levels. Also, we present a web application developed using the framework. Section 5 describes some issues for extending the framework.

2 Motivation

We believe that current systems tend to be either under-organised or over-organised. Users are either restricted by a fully-specified schema or left to their own information chaos. We therefore propose RIM as a solution.

As part of a larger project on *community information spaces*, we first want to implement an advanced email system using the RIM framework. This email system will provide users with a more flexible means of organising emails through classifications and associations. Further, it should be possible to store and organise information extracted from emails. Let us highlight some key problems of this application that can be solved using the RIM framework. Suppose, for example, that a professor is responsible for managing the submission and selection of papers for a conference. An initial schema for managing information might include a collection `Papers` containing objects of type `paper` with attributes `name`, identifying the name of the author of the paper and `file`, corresponding to the name of the file sent. As paper submissions are received by email, the professor will create the appropriate objects in the database. However, it may happen that the process does not go as smoothly as anticipated and some submissions are received which require special handling, e.g. a file that is damaged or does not correspond to the required format. With the RIM approach, the professor can simply and dynamically add additional attributes to the created object stating that another file is expected.

The use of heterogeneous collections and of operations performed on these collections can be motivated by the following example. In the context of previous application, suppose that the professor manages information about the persons participating at the conference by using the collection `Conference_Contacts` containing objects of type `contact`. Type `contact` has the attributes `name`, `address` and `email`. Further, information about friends can be managed by the use of collection `Friends` which mainly contains objects of type `friend`. Suppose type `friend` has the attributes `name`, `birthdate` and `interests`. Assume some persons from the conference become new friends of the professor, but for these persons the birthdate and their interests are not yet known. Using the RIM framework, it is possible to add to collection `Friends` objects of type `contact`, because all collections are heterogeneous. Operations over collections can be performed with or without type checking. For example, we can perform the intersection without type checking between collections `Friends` and `Conference_Contacts` to obtain the collection of persons from the conference that became friends of the user. Then, we can perform a selection with type checking over collection `Friends` to obtain all friends for which the birthday is a specified day.

There are different ways that users want to extract and organise information received as part of their everyday activities. Through simple drag and drop operations, they will want to move information between applications – such as email and calendar systems – and also to store it on their desktops. Easy management and retrieval of information dropped on the desktop requires an ability to organise that information, but without the need for pre-defined schemas. The information may be private, or part of a community information space in which users have different information spheres defining not only the scope of their information access, but also their information view.

We now go on to first describe the models on which RIM is based and then the initial framework that has so far been developed.

3 The eXtreme Design (XD) approach

In this section, we first describe the OM generic model which underlies our approach in terms of the basic constructs and operations required for semantic data modelling. We then go on to describe the eXtreme Design (XD) approach that we use for RIM. As we will show, XD is also based on the OM view of modelling and in fact has a metamodel expressed in OM as its core.

The **generic object model OM** is suitable for conceptual design because it can be used for all stages of the database development cycle [KNW98] and offers a rich set of concepts for classifying and typing.

For example, in a university we want to have an information system capable of managing `student` and `teacher` data. So, we can define the following type hierarchy in OM notation, as shown in Figure 1. We assign to a `person` properties such as `name`, `birthdate`, `address` and `email`. Additionally, we assign to a `student` the attribute `year_of_study` and the method `attends_courses` and to a `teacher` attributes `rank`, `salary` and `office` and the method teaches_courses. Further, we want to classify persons into students and teachers.

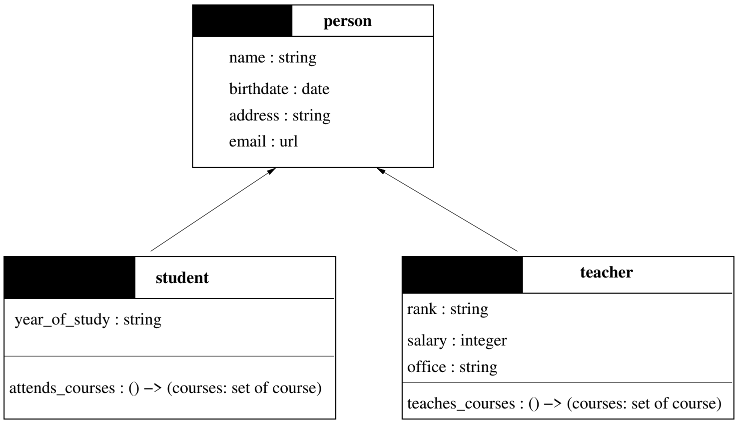

Figure 1: Typing hierarchy for University example

The OM model distinguishes between the *classification* of entities into categories according to general concepts of the application domain and the description of the representation of entities within the database in terms of *types* [Nor95].

In reality, a person can be classified as a student and as a teacher simultaneously. In this way, a person will play the role of a student and of a teacher at the same time. The OM model allows objects to have different types simultaneously. So, the same person object can be dressed with the types student and teacher. Such role modelling cannot be supported in most object-oriented data models.

In our example, as shown in Figure 2, role modelling is achieved by inserting objects into collections Persons, Students and Teachers assuming associated member types person, student and teacher respectively and it is possible that an object belongs to more than one collection at the same time. For instance, if an object is a member of the collection Students and Teachers simultaneously, it gains the attributes name, birthdate, address, email, year_of_study and the method attends_courses if it is viewed through collection Students and the attributes name, birthdate, address, email, rank, salary, office and the method teaches_courses if it is viewed through collection Teachers.

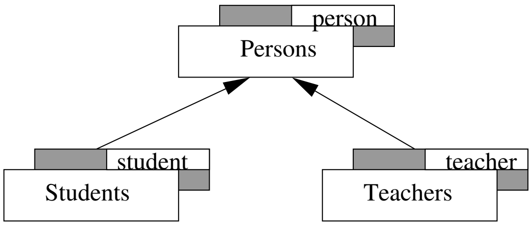

Figure 2: Classification

The OM model provides support for collections of objects and associations (binary collections). It specifies a set of operations over collections such as union, intersection, difference, cross product, flatten, selection, map, reduce and cardinality, and a set of operations over binary collections such as domain, range, inverse, compose and closure. Further, the OM model defines some integrity constraints such as subcollection, cover, disjoint, partition, intersection and cardinality constraints. For a full description of the OM model and its associated algebra see [Wür00].

The OM model can be used for all stages of database development. In the *conceptual database design* stage, we can express knowledge about objects, semantic groupings of objects and associations between objects — all using a very powerful graphical notation in which there are represented the collections of objects, the associations between these collections and the constraints [NW00]. During the *data model mapping* phase, the conceptual schema is transformed to the data model of the chosen DBMS by using Data Definition Language (DDL) statements. The population of the database is described in terms of Data Manipulation Language (DML) statements.

The **eXtreme Design (XD) approach** [Kob00] facilitates rapid information modelling (RIM) by making it possible to specify and implement concepts of the application domain on a very high level of abstraction. The eXtreme Design approach comprises the XD meta model and XD algebra and is supported by the XD framework.

The XD meta model is defined in term of the OM model, making it possible to easily customise the meta model by adding collections and associations. The set of object attributes and methods is not fixed and they can be added and removed at runtime.

Compared to other design methods such as the ones proposed in [Boo91, CY91, RBP91], we do not have to change the design specification and implementation if we want, for example, to introduce new object properties or additional relationships. Also, the third generation methods such as the Unified Modelling Language (UML) [UML] or Object-oriented Process, Environment and Notation (OPEN) [Hen97] provide techniques for categorising application domain objects, for defining relationships among objects and for specifying object properties. However, when introducing new requirements to a software system, often it is necessary to change and adapt the corresponding design specifications together with their implementations.

Figure 3 shows the XD meta model which is divided into three main parts corresponding to the activity of *classification*, *structuring* and *typing*.

The activity of **classification** corresponds to role modelling, i.e. the specification of the roles that the objects play in the application domain. `Objects` can be classified to be members of `Object Groups` as shown in Figure 3 by the association `are_member_of` between `Objects` and `Object Groups`. An `Object Group`, in turn, can be a member of other `Object Groups` as shown by the association `are_member_of` over `Object Groups`.

Structuring refers to defining relationships between objects. By the association `refer_to` between meta objects, we can express **hierarchical relationships** [DT93] such as *is_a*, *instance_of*, *part_of*, *membership_of* relationships and **non-hierarchical relationships** (*one-to-many* and *many-to-many* relationships). The `specified_by` association between the `refer_to` association and the `Objects` collection can be used to characterise the `refer_to` association by assigning facts [Kob00].

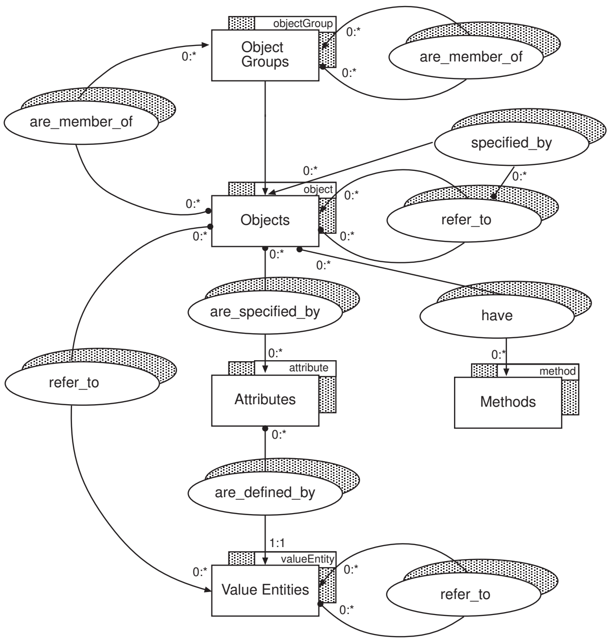

Figure 3: A schema in the OM model

Typing refers to methods, attributes and attribute values. Attributes are linked to the object by `are_specified_by` associations and methods are linked to the object by `have` associations. Value entities are associated to attributes by `are_defined_by` associations and to objects by `refer_to` associations. Value sets can be specified by `refer_to` associations between value entities.

The XD Algebra consists of algebraic operations for all three parts of the XD meta model, defining operations for object groups, for object relationships and facts and for object properties [Kob00].

4 Rapid Information Modelling

In what follows we will motivate the need for rapid information modelling by means of an example. Let us consider again the university example. Consider the `Students` collection. Each student object which is a member of this collection has associated properties such as `name`, `birthdate`, `address`, `email`, `year_of_study`, and `attends_courses`. Suppose that, after having the list of all students, we find out that 1% of the existing students not only study, but also work. Thus, for these students, we decide to keep some extra information about their job: `address_of_work` and `salary`.

In the case of the OM model, each collection has an associated type, and an object belonging to that collection is dressed with that type. A solution for the described problem would be to change the type `student` associated to the collection `Students` to reflect the extra information. In that case, the type `student` would have associated attributes `name`, `birthdate`, `address`, `email`, `year_of_study`, `address_of_work` and `salary`. But it does not make sense to have all these properties for the students who do not work. On the other hand, the type `student` associated to the collection cannot be changed at runtime. Another possibility would be to create a new collection `StudentsThatWork` with the associated type `studentThatWorks`. Type `studentThatWorks` has the attributes `address_of_work` and `salary`. So, a student that works can be accessed both from collection `Students` and then is viewed as being of type `student` and from collection `StudentsThatWork` and then is viewed as being of type `studentThatWorks`. But for this approach the schema has to be changed by creating the new collection `StudentsThatWork` and the new type `studentThatWorks`. Moreover, if we want to store different kinds of additional information for different students, we will end up with a lot of collections and types created only for representing some particular cases.

So, there are problems when we want to add some attributes to an object belonging to a collection without having to change the schema. The same problem arises when we want to delete some attributes of an object in a collection. For instance, when we insert a `student` object into the collection `Teachers`, we do not want to associate to it the attribute `office`.

We argue the necessity to add or remove attributes of an object at runtime and also to model heterogeneous collections. For adding and removing attributes of objects at runtime we have used the XD framework. The XD framework is a description at the level of objects and is based on the key notion that everything is an object. The RIM framework extends the XD framework by allowing a description at the level of collections. There were designed heterogeneous collections, i.e. collections in which the members have some common features, but not all of their properties are the same. Heterogeneous collections can be implemented by associating not only one, but more types to a collection.

In the RIM framework, we wanted to model the *data model* used at the client application level using the *meta model* from the server side.

According to the XD meta model, there are three levels corresponding to the three activities, as depicted in Figure 4: classification, typing and structuring. At the classification level we deal with collections and associations. At the typing level we allow an object to be dressed with more than one type. In the OM model implementation under Java, OMS Java, this was accomplished by constructing two classes: `OMObject` and `OMInstance` [NK00]. At the structuring level we deal with relations between objects. In what follows, we will explain in detail our approach for each of the three levels.

Classification level used in the 'OM-like' data model is implemented by means of collections and associations or binary collections and their algebra. We deal with heterogeneous collections, in this sense extending the OM model which allows only homogeneous collections.

A collection can contain objects of different types. Each object inside the collection has an associated type. Because a collection has different associated types, we can view the collection through any of these types. So, we can introduce the notion of *collection viewed through a type*.

We denote a collection $\mathtt{C}$ having members of type $\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}$ by $\mathtt{C}\{\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}\}$. Suppose A_i, $i = \overline{1,n}$, is the set of object instances of types $\mathtt{t_i}$ belonging to collection $\mathtt{C}$. Then, $\mathtt{C}\{\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}\} = A_1 \cup A_2 \cup \ldots \cup A_n$. $\mathtt{C}$ *viewed through the*

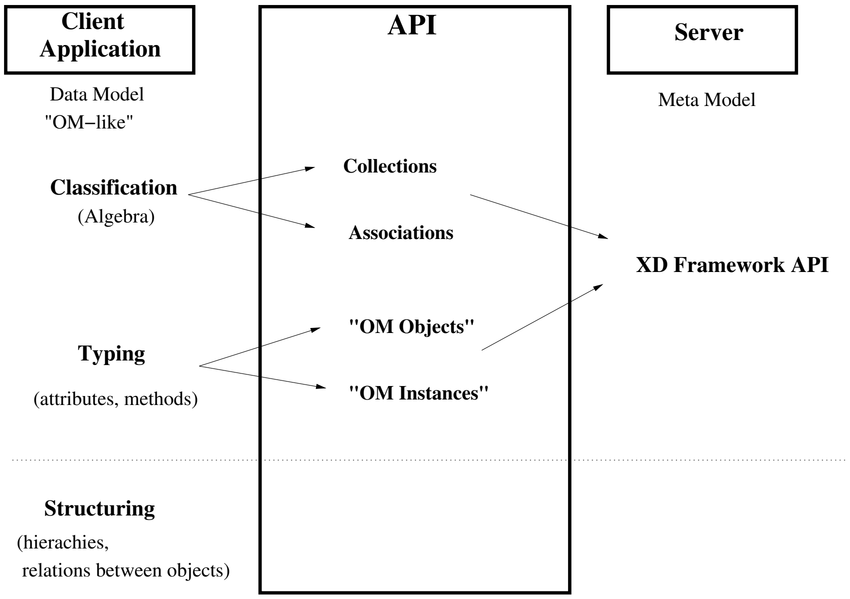

Figure 4: Architecture

type $\mathtt{t_i}$, denoted by $\mathtt{C}(\mathtt{t_i})$ will contain the set of objects instances of type $\mathtt{t_i}$, i.e. $\mathtt{C}(\mathtt{t_i}) = A_i$.

We also have introduced some graphical notations for representing a collection together with its members and for representing an object instance of a type together with its attributes, as shown in Figure 5. Note that the object can have some missing or extra attributes compared to the attributes of the type. When representing it like this, there will be listed only the existing attributes of the object which belong also to the type.

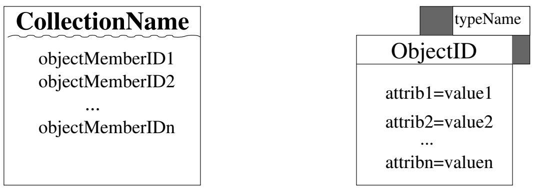

Figure 5: Collection and Object Representation

We have introduced the notion of *base type of a collection*. For a collection we can explicitly specify the base type, according to the semantics of that collection. For instance, for collection `Students` the base type can be specified to be `student`, for collection `Teachers` the type `teacher` and so on. But, if no base type is associated to a collection, it can be established to be the type of the majority of members of the collection. For a collection $\mathtt{C}\{\mathtt{t_1}, \ldots, \mathtt{t_n}\}$, if the base type is $\mathtt{t_i}$, then it will be denoted by $\mathtt{C}[\mathtt{t_i}]$.

For the heterogeneous collections we have defined a set of operations: cardinality, union, intersection, difference, selection, map and reduce.

These operations can be realised in two ways:

- **without type checking** meaning that the operations will not take into account the types of the members of the collections. The operations are evaluated in the same way as operations over two sets.
- **with type checking** meaning that the operations will take into account the types of the members of the collections, that means the collections are viewed through a certain type.

We will use an example for illustrating how a union operation can be performed with or without type checking. Consider the collection `Students` containing as members the objects given by their object identifiers: `OID1`, `OID2` and `OID3`. Objects represented by `OID1` and `OID2` are dressed with type `student` and the object represented by `OID3` is dressed with type `person`. Suppose that the base type of collection `Students` is type `student`, denoted by `Students[student]`. Further, consider the collection `Teachers` having as members the objects given by their object identifiers: `OID4`, `OID5` and `OID1`. Objects represented by `OID4`, `OID5` are dressed with type `teacher` and the object represented by `OID1` is dressed with type `student`. Suppose that the base type of collection `Teachers` is type `teacher`, denoted by `Teachers[teacher]`.

Let us consider the union operation between collections `Students` and `Teach-ers`. When performing *union without type checking* we obtain

$$\mathtt{Students} \underset{\substack{\text{without type}\\ \text{checking}}}{\cup} \mathtt{Teachers} = \{\text{OID1, OID2, OID3, OID4, OID5}\}$$

The result will be a collection which will not have implicitly associated a base type, but it can be established to be the type of the majority of members of the collection. In this case there are 2 members of type `student`, 2 members of type `teacher`, and 1 member of type `person`. So, the base can be either `student` or `teacher`.

In the case of *union with type checking*, we will consider only the members of the two collections which are instances of the base types of the collections. The result collection will have as base type the least common supertype of the base types of the collections. The least common supertype has as properties the intersection of properties of the type `student` and of the type `teacher`. So, the least common supertype of type `student` and `teacher` is type `person`. The result of the union with type checking of collections `Students` and `Teachers` is

$$\mathtt{Students}\,[\mathtt{student}] \underset{\substack{\text{with type}\\ \text{checking}}}{\cup} \mathtt{Teachers}\,[\mathtt{teacher}] =$$
$$= \{\text{OID1, OID2}\} \cup \{\text{OID4, OID5}\} = \{\text{OID1, OID2, OID4, OID5}\}$$

The object members of the result collection are dressed with type `person`, but they did not lose the additional properties corresponding to type `student` for objects given by `OID1` and `OID2` and respectively to type `teacher` for objects given by `OID4` and `OID5`.

At the **typing** level, as in the OM model, we allow an object to have different types simultaneously. We have seen that in the eXtreme Design meta model the set of object attributes and methods is not fixed. Suppose we create an object and dress it with a certain type. What happens if we remove or add attributes to the object? In the case of strong typing, the object would no longer be an instance of that type. So, typing should allow some flexibility.

We say that an object can be considered to be dressed with a type even if the set of attributes of the object is different than the set of attributes of the type. Of course, this difference cannot exceed some limits.

An object can be dressed with a certain type if one of the following conditions is fulfilled:

1. the object has a **number of common attributes** with the type
2. the object has **at most Δ extra attributes** and **at most Δ missing attributes** compared to the attributes of the type, Δ being specified
3. the ratio between the number of common attributes and the number of different attributes must be greater than a given limit.

$$\frac{\text{no. of common attribs}}{\text{no. of missing attribs + no. of extra attribs}} \geq \text{percentage},$$

where percentage is given.

The **structuring** level refers to specifying object relationships. For example, in our framework we have developed subcollection relationships.

Also, the framework offers some functions in which information is analysed and schema and data definition files generated in terms of the DDL (Data Definition Language) and DML (Data Manipulation Language) associated with the OM model. These can then be loaded into a particular data management system based on the OM model such as OMS Java [NK00].

We have developed a web application which can be considered as a prototype for RIM. For the connection between the browser and our application we have used Java Servlets [Mos99]. The application allows the user to create objects of different types, to update objects including the possibilities of adding and removing new attributes, to create new types and collections, and to insert or remove objects into collections. Also, the user can choose one of the typing constraints we have mentioned previously, according to which an object is determined to be instance of a certain type. According to the type an object is dressed with, the user can find out the most suitable collection for containing that object. The user can perform algebra operations over collections, either by typing a query in the Algebraic Query Language (AQL) [NW00] or by being guided to select the parameters necessary for performing different operations.

5 Future Work

In this section we present some issues for future extensions to the framework.

The framework can extend the typing constraints to refer not only to attributes, but also to methods. We refer to the attributes and methods of an object as the properties of that object. Then, the typing conditions will be modified to refer to properties instead of attributes. A more general case is to take into account not only the number of properties, but also some specified properties.

We also propose to develop a full algebra for binary collections. This can be done with some modifications to the framework in order that all operations on collections apply also to binary collections. From the specific operations for binary collections, there were implemented only *domain* and *range*. But, there can be implemented also *domain restriction* and *subtraction*, *range restriction* and *subtraction*, *inverse*, *composition*, *nest*, *unnest*, *division* and *closure*.

The *subcollection constraint* has been implemented, but other forms of classification constraint such as the *cover constraint*, *disjoint constraint*, *partition constraint* and *intersect constraint* could also be implemented.

Currently, a collection must be of bulk type *set*. The framework can be extended for supporting also bulk types *bag*, *ranking* and *sequence* behaviour which are supported in the OM model.

After the schema has been generated, it would be possible to automatically create a *hierarchy of types* by analysing, not only the attributes of the types, but also information concerning objects and collections. If two types have some common attributes, it does not imply that the two types are in a subtype relationship. We have to analyse whether there exist objects instances of both types or if there exist collections containing members of both types, because a collection means a grouping of objects having some common features.

Because we are using a metamodelling approach, the additional flexibility is traded for the cost of less structure. For example, what happens when querying on an object for an attribute that it was already removed? For this problem we can apply the exception handling mechanism discussed in [Bor85a]. In this paper, an Information System is defined to be a computer system maintaining knowledge about some aspect of the world and consisting of two main parts: a database of persistent facts and a collection of application programs ('transactions') which are run against the database in order to retrieve or update information on it. An exceptional object is created when, for example, a transaction tries to access a missing attribute or one of the constraints is violated. When exceptional facts are tried to be accommodated, some of the transactions will be semantically incorrect or incorrectly typed. An essentially 'demand driven' technique [Bor85b] for dealing with exceptional values is to allow transactions which signal or propagate exceptions either to be 'backtracked', so that all side effects are undone, or to be 'resumed'. The resume possibility means that the exceptional value is replaced by another value for the particular case. For instance, in case of missing attributes, we can provide null values to indicate lack of information. A problem for which we have not found yet a very suitable solution is the following one. Suppose a method accesses an existent attribute of an object. Further, suppose this attribute is removed and then there is added another attribute with the same name but different type. The method relies on the fact that the attribute is of the older type. The exception mechanism is not yet implemented in the framework.

As future work, the desktop vision application we have described in section 2 will be developed. The existing RIM framework will be extended and we will develop some other applications using this framework to demonstrate its generality.

For instance, we want to investigate its suitability for the information management of web sites.

References

[Boo91] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings Publishing, 1991.

[Bor85a] A. Borgida. Thoughts on accommodating exceptions to (type) constraints in Information Systems. In *Persistence and Data Types Papers for the Appin Workshop*, August 1985.

[Bor85b] A. Borgida. Language features for flexible handling of exceptions in Information Systems. In *ACM Trans. on Database Systems*, December 1985.

[CY91] P. Coad and E. Yourdan. *Object-Oriented Design*. Prentice-Hall, 1991.

[DT93] T. Dillon and P.L. Tan. *Object-Oriented Conceptual Modelling*. Prentice Hall, 1993

[Hen97] B. Henderson-Sellers. *OPEN: Object-oriented Process,Environment and Notation, The first full lifecycle, third generation OO method*. CRC Press, 1997.

[Kob00] A. Kobler. *The eXtreme Design Approach*. Draft Thesis, 2000

[KNW98] A. Kobler, M.C. Norrie and A. Würgler. OMS Approach to Database Development through Rapid Prototyping. In *Proc. 8th Workshop on Information Technologies and Systems (WITS'98)*, Helsinki, Finland, December 1998

[Mos99] K. Moss. *Java Servlets: second edition*. McGraw-Hill, 1999

[Nor95] M.C. Norrie. Distinguishing Typing and Classification in Object Data Models. In *Information Modelling and Knowledge Bases*, vol VI, ch.25 IOS, 1995 (originally appeared in Proc. European- Japanese Seminar on Information and Knowledge Modelling, Stockholm, Sweden, June 1994)

[NK00] M.C. Norrie and A. Kobler. *OMS Java Object-Oriented Framework and Data Management System*. Institute for Information Systems, ETH Zurich, May 2000

[NW00] M.C. Norrie and A. Würgler. *OMS Pro Introductory Tutorial*. Institute for Information Systems, ETH Zurich, March 2000

[RBP91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.

[UML] UML Resource Center. http://www.rational.com/uml/index.jsp. *OMG Unified Modelling Language Specification Version 1.3*, 1999

[Wür00] A.P. Würgler. *OMS Development Framework: Rapid Prototyping for Object-Oriented Databases*. Phd thesis, Departement of Computer Science, ETH, CH-8092 Zurich, Switzerland, 2000