Application Modelling in Heterogeneous Environments using an Object Specification Language*†

Gunter Saake, Ralf Jungclaus, Thorsten Hartmann
Abt. Datenbanken, Techn. Universität Braunschweig
Postfach 3329, W-3300 Braunschweig, FRG
E-mail: {saake|jungclau|hartmann}@idb.cs.tu-bs.de

Abstract

In this paper, we propose an object-oriented logical formalism to conceptionally model applications in an interoperable environment. Such an environment consists of heterogeneous and autonomous local (database) systems. Applications in such an environment use several resources and services. Their conceptual modelling involves re-specification of existing systems in terms of homogeneous views, modelling of behavior and system dynamics, modelling of logically distributed components in an open environment and the modeling of communication relationships and dependencies between components. We introduce a formal object-oriented language capable of dealing with these requirements and illustrate its use to model applications in an interoperable environment.

1 Introduction

We are currently facing a situation where more and more applications are aiming at integrating and using data and services of local (database) systems [3]. Information has become a key factor in industry, thus it becomes essential to establish relationships between already existing resources and services (e.g. in CIM and Office Systems). *Heterogeneous (or Federated) Systems* [29, 3] have been promoted to provide a basis for the integration of data and database systems. The capability of systems to cooperate in such environments is called *interoperability*.

Our objective in this paper is to contribute to a *conceptual* approach to integrate distributed and heterogeneous resources and services through global applications. Such applications are based on logically distributed, heterogeneously structured and often semantically overlapping (*semantic compatibility* [25]) data and services. On the conceptual level, we have to be concerned about finding out which data or service is relevant, where it is located and how information can be structurally and dynamically related. It is obvious that global applications on top of several (remote) databases are becoming more and more important in the light of distributed heterogeneous systems in one organization [31].

It is important to model global applications *independently* of implementation and concrete data model issues as far as possible. Only this approach makes local systems compatible [23, 3] and allows us to deal with semantic reconciliation and semantic-based integration in a formal way. Furthermore, applications are *dynamic* – they model e.g. office or business procedures or manufacturing processes or even a combination of them. Therefore, a conceptual formalism must be able to model dynamics and behavior.

A special point is that an interoperable environment must be regarded as being *open* – it must be extensible by new local systems and applications, the local systems must remain autonomous as far as possible. Thus, we must also be able to model on a logical level how federated components are interconnected and how communication takes place between them. Furthermore, we should be able to model *views* or interfaces on components [17].

The approach described in this paper introduces an object-oriented logical formalism that allows to both *re-modelling* local databases *and services* in terms of a global canonical model and to conceptually specify global applications on top of local databases and even other types of local systems. The ideas of object-orientation nicely reflect the concept of a client-server relationship between applications and underly-

*This work was partly supported by CEC under ESPRIT-III BRA WG 6071 IS-CORE (Information Systems – COrrectness and REusability) and by Deutsche Forschungsgemeinschaft under grant no. Sa 465/1-2.

†To appear: Proc. Int. Conf. on Intelligent and Cooperative Information Systems ICICIS'93, IEEE CS Press 1993

ing database systems, it *integrates* structural and dynamic modelling of components, it allows us to *encapsulate* components and supports extensibility of federations through interfaces and communication. The idea is to model the application as objects that are embedded in an environment of interacting objects that represent the data repositories and their services. This view is similar to the one described in [4, 3], although the realization is different.

The paper is organized as follows: In the next section, we list and discuss requirements for the conceptual modelling of global applications in an interoperable environment. In Section 3, we briefly and informally introduce a formal object-oriented framework. In the following Section 4, we describe features of a logic-based object-oriented modelling language (`TROLL` [15]) that we think are important in an interoperable environment. In Section 5, we sketch a method on how to model applications over existing system components. Section 6 then illustrates how our approach is applied to a toy example of the CIM field before we give some conclusions.

2 Requirements for Modelling Interoperable Applications

In general, we have some requirements for conceptual modelling of applications which are more or less independent of the specific situation of interoperable environments. We will first summarize these properties before we come to specific effects caused by the aim of having applications in interoperable environments.

- A conceptual model should *abstract* from implementation details both concerning data structures and computation functions.
- A language for conceptual modelling needs a *formal semantics* and a corresponding *specification logic* to enable consistency checking and verification of implementations.
- To model *applications* instead of pure database structure, we need an *integrated formalism* to model structure *and* behavior [10, 18, 20].

Besides these general requirements, we have a special situation if we want to specify application processes in an interoperable environment:

- Typically, we will not build a system from scratch but have to integrate existing subsystems in our application. This leads to two effects:
  - We have to *'remodel'* existing systems, i.e. we have to formally describe existing already implemented systems (as far as possible).
  - As a consequence, typical top-down modelling approaches like functional decomposition alone are not sufficient — we need *bottom-up techniques* as they are for example offered by object-oriented design approaches, too.
- Components of our environment may be *heterogeneous* with respect to the data model being used and the services being offered. To model an application over heterogeneous subsystems, we need to formalize *homogeneous views over heterogeneous databases and services.*
- Subsystems may have different degrees of *autonomy and encapsulation.* Therefore, we have to express e.g. different communication patterns and exception handling. In many cases, we do not know the semantics of services offered by a subsystem because of encapsulation which means that we are not able to completely remodel it.
- A typical situation in interoperable and open environments is that we do not have one complete and fixed global view of our system — and that such a view cannot exist because of extensions and integration of new systems during the lifetime of an application. This effect requires that the specification formalism being used supports *modularity and extensibility.*
- Application processes are inherently *logically distributed computations* not necessarily following the classical transaction paradigm, in particular there may be cooperation between transactions.

During the design of an application in such a heterogeneous environment of systems having different degrees of autonomy and encapsulation, we have to formally integrate in our conceptual model different kinds of system components affected by the application process:

- Parts of the complete system will be newly constructed during the design process. In this case, we are free to choose the description framework and we have to completely specify the system structure and behavior.
- Other system parts do already exist, and we have a more or less formal description of their semantics. A typical example is an existing relational

database to be used by an application. Since we want to use their properties e.g. for consistency checking, such subsystems have to be re-specified in the description formalism used for the conceptual modelling of the application task.

- Other components are completely encapsulated, because they have a high degree of autonomy and allow a restricted access by predefined services only. In many cases, all we know about the semantics of such a system is the signature of services offered to the outside.

- In a global application, we have to deal with dependencies and communication between components. That means that we have to have modelling concepts to formalize such connections.

- Global applications may be composed from tasks or procedures. A modelling formalism has to be capable of describing such tasks.

We think that a formal object-oriented approach to conceptual modelling is the most promising way to capture these strong requirements. We will present features of such an approach and sketch how it can be used to model applications in an interoperable environment.

3 An Object-Oriented Framework

It seems to be more or less accepted that the object-oriented approach is suitable for the design of information systems [2]. Objects encapsulating structure and behavior can be regarded as basic building blocks of system descriptions [24]. A number of less formal object-oriented modelling approaches are becoming more and more popular in systems design [21].

In our approach, an object is an observable communicating process encapsulated by an access interface. Observations consist of reading its attribute values, object changes over time are driven by occurrences of local events. An event occurrence may be initiated by the object itself (active object) or as a result of a communication with another object (event calling). Both the values of attributes and the possible parameters of events are *data values* in the sense of abstract data type values.

The basis for a formalization of such an object concept are values structured using the concept of abstract data types (ADTs) [6]. Data values are elements of the carrier set of some ADT. An ADT is defined by a carrier set for data values along with typed functions on this set.

For the formalization of object signatures, we use the notion of a signature as known from the ADT framework. An object signature consists of attribute and event symbols. Attributes are null-ary functions into a certain domain (a data type), events are functions with parameters and a special event sort as domain. The object signature defines the alphabet for an object specification in a formal framework.

The next step is the formalization of object evolutions using linear processes in a process framework. A possible choice is to adopt the life cycle model [16] which defines a process as the set of possible snapshots of events. A snapshot is a set of concurrently occurring events. Special attention must be paid to events that create and destroy an object. Therefore we require the first snapshot in a life cycle to include at least one *birth* event. If the life cycle is finite, a *death* event is included in the last snapshot. Note that death events are not required, thus there may be objects that exist infinitely.

The last thing to be formalized for the specification of simple objects is the *observation of current object properties.* This is achieved by introducing an observation structure associating attribute values with each reachable state of an object. A reachable object state is given by a finite prefix of a possible object life cycle. The observation structure can be described by a mapping of object states (life cycle prefixes) to attribute-value relations [30].

Up to now we have only sketched a formalization of single objects. Another topic is the definition of object composition. Component relations between objects are modeled using structure preserving mappings between objects, the so-called object morphisms [7], which can be be compared to process combination known from process theory. As a special case we have inclusion morphisms where the mapping is supposed to be injective. Inclusion morphisms can be used to explain the *embedding of subobjects* into objects, i.e. aggregation of objects. The embedded objects are regarded as subprocesses in the enclosing composite object, where only events local to the subobject may have effects on attribute observations (of the subobject attributes).

The step from single objects to sets of objects is done by introducing an identification mechanism for object instances. Again we use ADTs to describe object identifiers. A class type defines a set of identifiers along with a prototype object model (an object template). An object class then consists of a class type and a set of object instances together with a mapping from a set of object identifiers to actual object instances.

The basic mechanism to support communication between objects is event calling. When an event e_1 calls another event e_2, then whenever e_1 occurs, e_2 must also occur simultaneously (but not vice versa). Thus, event calling is an asymmetric synchronization on event occurrences inside an object, which itself can be composed from several objects using object inclusion. Because of the definition of object embedding, calling is the only way an embedded object can be manipulated by the embedding one.

A more elaborated formalization of these ideas can be found in [16, 26, 13]. For recent developments towards a more sophisticated semantic domain for object systems see [7, 8].

4 Language Features of TROLL

A specification language to be used in interoperable environments must support the description of the conceptual schema according to different viewpoints. Thus, it must offer language features to describe different types of subsystems. One dimension is the remodelling of existing systems and the definition of interfaces to existing systems if the semantics of operations is not completely known. Another dimension is the modelling of components that make up the core of the new application. In order to integrate the different subsystems we have to define suitable relationships and communication channels between them.

We will proceed with the basic language constructs to describe single objects, object classes, the composition of objects to complex objects, interfaces to objects and relationships between objects.

In TROLL the structure and behavior of an object is specified by means of a *template*, the object itself by a proper name together with the template.

Example 4.1 See for example a product object representing a part in a manufacturing company :

```
object product
template
   attributes
      Name:  string;
      Price:  money;
      QOH: integer;
   constraints
      QOH >= 0 and Price > 0;
   events
      birth CreatePart(...);
      UpdateQOH(integer);
      UpdatePrice(real);
      ...
   valuation
      variables factor:real; n:integer;
      [UpdateQOH(n)] QOH = n;
      [UpdatePrice(factor)]
            Price = Price * factor;
      ...
   permissions
      variables factor:real;
      { factor >= 0.8 and factor <= 1.2 }
            UpdatePrice(factor);
      ...
end object product;
```

The attributes and events make up the (local) signature of an object. The signature makes up the alphabet underlying the specification of objects.

The attributes define the observable properties whereas the events define the behavior interface of objects. In the `constraints` section we may describe conditions that must be fulfilled for all possible attribute observations. Here we specified that the `QOH` of products as well as their price may not be negative. In general we allow for constraints formulated in a future tense temporal logic based on [22, 19].

Events describe the state changing operations of an object. One special event, `CreatePart`, is qualified as the *birth* event of the example object. The birth event creates an object and must therefore be the first operation in the object life cycle. For more specific state changes we allow for parameterized events. In the example the event `UpdateQOH` has an integer parameter. Parameters may be used for data interchange between objects during communication and for defining the effects of events on observations.

Effects on attribute values are specified by means of valuation expressions. A valuation expression describes the effect of an event occurrence (possibly instantiated with appropriate data type values) on attributes in terms of data terms describing the new attribute value. In the example after the event `UpdateQOH(n)` instantiated with an integer value `n` occurred the attribute `QOH` has value `n`. Note that the formula denoting the new value is evaluated in the object state before the event occurs.

For dynamic objects we must not only be able to describe their behavior in terms of state changing operations but we have to describe also their *possible* behavior in terms of allowed object life cycles. In TROLL this goal is achieved with preconditions on events (*permissions*), conditions to be fulfilled for complete life cycles (*obligations*) or using an explicit process definition language based on CSP [14]. The example only states a rather artificial precondition allowing price changes only in a small interval. In general we may refer to the history of an object by means of a past tense temporal logic for preconditions. More complex life cycle descriptions via explicit process definitions are given in the next section.

Besides the description of single objects, we may specify classes of objects by introducing an identification mechanism for instances. The identification mechanism describes the set of external identifiers of instances.

Example 4.2 For product objects we may specify a class :

```
object class PRODUCT
   identification PartNo:  integer;
   template product;
end object class PRODUCT;
```

consisting of part number IDs and templates adopted from the previously defined product objects. The external identifiers are mapped to immutable internal surrogates. []

For modelling a system of objects it is not sufficient to specify the objects in isolation. Some objects in a system may be closely connected and form *complex* objects, other ones are only related by communication channels. In `TROLL` the former are specified with an inclusion mechanism, the latter are described with relationships.

Inclusion is the basic language feature to relate separately specified objects. First, with including an object into another object, the (local) signature of the latter is enriched with the signature of the former, i.e. we may use attribute and event symbols of both objects. Moreover, the object semantics of the embedded object is incorporated into the embedding object. That is, we may refer to the actual properties of both objects in the context of the aggregation.

On the language level composite objects are specified as components. Components are either single object instances or lists or sets of instances.

Example 4.3 A store of products may thus be specified in the following way :

```
object STORE
template
   components
      Prods:SET(PRODUCT);
   ...
end object STORE;
```

With the specification of components we have generated events and attributes to update and to observe the composition. In case of set valued components these are events like `Prods.INSERT(|PRODUCT|)` to include product objects and attributes or `Prods.ID_SET` denoting the set of object IDs of product objects currently in the composition. []

Communication inside such complex objects is achieved by means of calling. When events e_1 and e_2 are related by calling (e_1 >> e_2), then the occurrence of e_1 forces e_2 to occur simultaneously. Thus, calling can be characterized as synchronous communication between the components of a complex object. For examples see the next section.

This simple pattern of communication may be used for objects that are closely related. In specifying object systems we often encounter the fact that we must also loosely connect objects. Consider e.g. the (representation of) the human user of a product object may want to force the event `UpdatePrice` to occur. Here it is unnatural to specify the product as a part of the user object. For this purpose we introduced the relationship construct. Since communication is not buried inside objects, this approach also supports reusing object specifications.

Example 4.4 Suppose we specified an object class `USER` with an event `SetNewPrice`. With a calling relationship between the user and the product, we achieve the necessary communication.

```
relationship UserProductInteraction
   between USER, PRODUCT;
calling
   variables f:real; p:|PRODUCT|;
   USER.SetNewPrice(p,f) >>
            PRODUCT(p).UpdatePrice(f);
   ...
end relationship UserProductInteraction;
```

Note that `|PRODUCT|` denotes the identification space implicitly introduced with the specification of the class `PRODUCT`. []

Relationships may be refined to communication channels in the following design phase. There, communication channel objects to describe relationships are introduced. These objects are included into the communicating objects, thus gluing together related objects at the specified points in their life cycles [12]. Extending event calling, we may also specify called *processes* defining complex communication patterns between objects. Processes may be formulated using sequential composition, choice and recursion. For an example see the next section.

As mentioned in Section 2, we sometimes have to model system parts that are not completely known with respect to their semantics or parts where only a restricted set of properties is relevant for the new application. Therefore we introduce *interfaces*. In general, an interface defines a restricted view on objects, i.e. some object properties are encapsulated and may not be used from the outside.

For the modelling of interoperable applications, interfaces may be used to define system components where only parts of the operation and observation signature is known. An interface thus lists the attribute and event signature.

Example 4.5 As a restricted view of `product` objects we may define an interface where only the `Name` and `Price` attributes are visible and only the event `UpdatePrice` may be used from the outside.

```
interface RestrictedProduct
   encapsulating PRODUCT;
template
   attributes
      Name:string;
      Price:money;
   events
      UpdatePrice(real);
end interface RestrictedProduct;
```

In general interfaces may also be used to encapsulate design steps following the specification phase. This will be especially useful for newly created (not remodelled) components of the application.

In this section we have given a survey of a simplified `TROLL` version where only few basic language features needed for the rest of the paper are introduced. For a thorough definition of `TROLL` see for example [15]. In [13] we describe complex objects and communication, in [26] we describe interfaces, and [28] includes in particular ideas for modularization of object societies.

5 Towards an Application Modelling Method

In this section, we sketch a way how to model applications over a (possibly heterogeneous) environment. The basic idea is to make the heterogeneity of existing (sub-)systems *transparent* to applications.

We start with the *identification of affected (sub-)systems*. This can be accomplished by identifying the sources of information the new application deals with and the *services* that can be used to accomplish its task. Therefore we strongly support the point that each organization should strive to produce a catalog of information sources and services of existing systems.

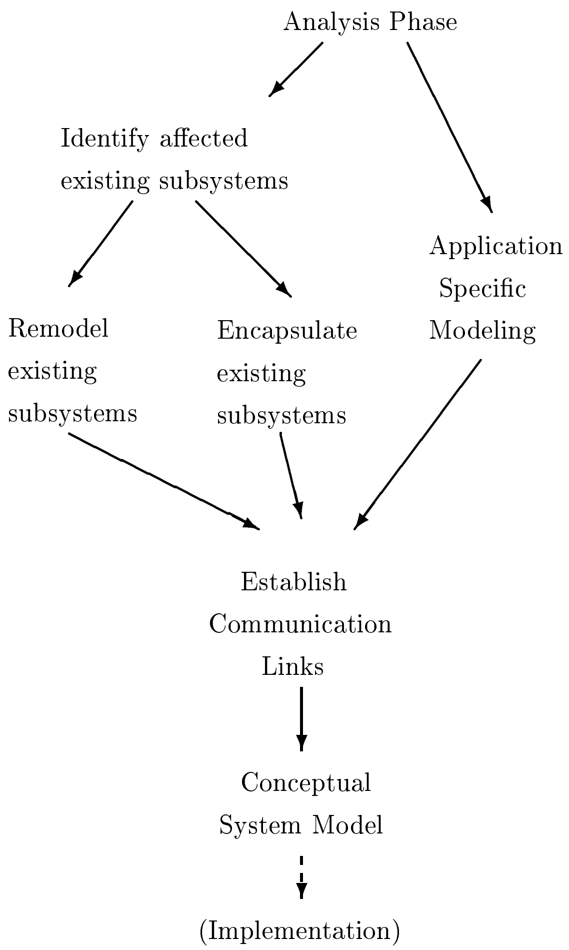

Figure 1: Application modelling method

The (sub-)systems then have to be *homogenized*. This is necessary to advance towards interoperability of heterogeneous subsystems [3]. We propose to *remodel* existing subsystems using the `TROLL`-language described in the previous section. The language provides features to model structural and dynamic properties of system components and thus can be used to model information sources and services in an integrated way. We think it is reasonable to distinguish encapsulation without knowing the semantics of services (*syntactic encapsulation*) and remodelling systems with known semantics.

On the other hand, we have to model the application. As argued before, we may be using `TROLL` also for this task. TROLL is particularly suited to model static and dynamic aspects of applications on a very high level of abstraction. Here we specify *new system components*.

By establishing *communication links* between these new system components and encapsulated subsystems we construct a *conceptual system model* of the complete application.

In the following stage of *system design*, we refine the new components and *represent them through existing components*. This is the most difficult part during the development process. The use of the *same formalism* for modelling the application and the existing services, however, enables the *verification* of implementation steps because we are able to check whether the implementation chosen fulfills what has been specified on a more abstract level. The use of TROLL makes this analysis step possible both for structural and behavioral properties. First approaches in a similar domain are reported in [9].

After a number of implementation steps we have refined our model such that it can be transformed into a (distributed) application over the services and information sources (e.g. databases).

A sketch of this method is illustrated by Figure 1.

6 A Toy Example

The usefulness of object-oriented specification techniques for conceptual modelling can be demonstrated using a small toy example, which shows relevant properties of typical applications in heterogeneous environments.

The example specification models part of a CIM application, namely the assembly process for composite parts. Of course, we cannot model a realistic application in this paper but have to abstract from modelling details to show general principles.

We assume to have three components affected by the assembly process — a catalog of parts stored in a relational database, a system performing the assembly (on top of an OODBMS to store hierarchical assembly plans), and a system located at a supplier company producing base parts where we only have restricted access via remote transaction calling for ordering parts.

Our aim is to model part of the assembly process of composite parts, where the catalog relation is used (and updated) and if necessary an order for parts is send to the supplier.

Example 6.1 We start with 're-specifying' the relational catalog by defining an object class representing the data stored in the relation.

```
object class CATALOG
   identification
      PartNo:  integer;
template
   attributes
      Name:  string;
      Price:  money;
      QOH: integer;
      CriticalQOH: integer;
      QuantityToOrder:  integer;
      Ordered:  bool;
   events
      birth CreatePartEntry(...);
      UpdateQOH(integer);
      UpdateOrdered(bool);
      ...
   valuation
      variables n:integer;
      [UpdateQOH(n)] QOH = n;
      ...
end object class CATALOG;
```

We assume that this object class specification is automatically compiled from a relational database schema — we have as events therefore no application specific events but generated update events for each attribute with straightforward valuation. In most cases, we would encapsulate such a generated interface by an external view to avoid misuse and integrity violation [26, 27]. []

In this example, we have interpreted a relation by an object *class* where objects correspond to tuples — another possible representation is to interpret a relation as one single object, which is more natural in certain applications. Such an example is given in [27].

For simplicity, we assume to have only one company where we can order parts using remote transaction call. The services of the company offered via an electronic network are modeled by a single object. The modification to several suppliers can be done by specification of an object class instead of a single object.

Example 6.2 The object supplier basically offers a collection of services to be called from the outside. Since we have no complete information on the semantics of the offered services, we model it as an object *interface* instead of an explicit object.

```
interface supplier external
template
   events
      RecieveOrder(in partno:  integer,
                   in quantity:  integer,
```

```
                    out okay?:  bool);
      ...
end interface supplier;
```

We show only one service in this interface — the event `ReceiveOrder` having two input parameters and one boolean parameter being set to `true` after a successful execution of the called service. []

The keyword `external` declares the interface to have an 'unknown' semantics, because otherwise an analysis of the specification would detect an incomplete specification.

The main application is modelled using active objects representing hierarchical assembly plans. Please note that this part of the modelling can be compiled into objects of an OODBMS in a straightforward manner.

Example 6.3 A part can be a basic part or it is composed from several subparts. Basic parts have a `CatalogEntry` attribute referencing a tuple in the catalog relation. Please remember that `PART` objects correspond to part descriptions in an assembly plan, not to 'real' parts in the production process.

```
object class PART
   identification PartName:  string;
template
   attributes
      constant IsBasicPart:  bool;
      CatalogEntry:  |CATALOG|;
      ...
   events
      birth CreatePart;
      Assemble;
      TakeFromStore;
      OrderPart;
      FinishAssembly;
      ...
   components
      SubParts:  SET(PART);
   constraints
      IsBasicPart => not undefined(CatalogEntry);
   ...
   process calling
      Assemble >> AssemblyProcess
      with process AssemblyProcess =
         foreach p in SubParts.ID_SET do
            SubParts(p).Assemble
         od
         -> { IsBasicPart = true } TakeFromStore
         -> { foreach p in SubParts.ID_SET :
              sometime(
               after(SubParts(p).FinishAssembly))
            } FinishAssembly
      end process;
end object class PART;
```

Please note that we have several attributes and manipulation events automatically generated with a components declaration [15, 13]. In our example, the generated set-valued attribute `SubParts.ID_SET` contains the object identifiers of subparts.

The active process `AssemblyProcess` of the main part recursively starts subprocesses `AssemblyProcess` indirectly via enforcing the events `SubParts(p).Assemble` at each included subpart — this may show the power of using process calling in complex objects. []

In this specification example we have used a language feature being an extension of the first `TROLL` language definition, namely *process calling* to describe the assembly process triggered by the `Assemble` event. This is not an extension of the expressive power of `TROLL`, because we can express the same effect using active phases of objects.

The last thing to do is to *connect the different (sub-)systems* declared until now. In `TROLL`, we can do this using two basic formalisms: building complex objects with internal communication, or declaring communication *relationships* playing the role of abstract communication channels. We will use the latter one.

Example 6.4 In our example, we need two relationships to describe the necessary communication.

```
relationship PartCatalog
   between PART P, CATALOG C
   where C.oid = P.CatalogEntry;
calling
   P.TakeFromStore >>
         C.UpdateQOH(C.QOH - 1);
end relationship PartCatalog;
```

The `where` clause ensures that the quantity on hand of the correct catalog entry is decreased.

```
relationship CatalogSupplier
      between CATALOG C, supplier
calling
   variables b:bool; n:integer;
   { QOH - 1 < CriticalQOH } ==>
      C.UpdateQOH >>
         supplier.ReceiveOrder(C.PartNo,
                     C.QuantityToOrder,b);
   supplier.ReceiveOrder(C.PartNo,n,true) >>
         C.UpdateOrdered(true);
end relationship CatalogSupplier;
```

Note, that a more elaborated specification would express this communication by a complex object realizing a script. □

Interaction relationships basically declare two things: They establish explicit communication channels and specify the control flow in terms of event calling. In most cases, event calling can be seen as an abstraction of the well-known trigger concept in the database community. Event calling realizes a simple synchronous communication concept for coupling several system components.

If we need a more sophisticated application process (similar to sagas or engineering transactions in the database context [11, 32]), we have to declare transient complex objects, where the application process is declared as an active internal process of an object and this object can be destroyed after performing the application process.

7 Conclusions

In this paper, we concentrated on the modelling of applications in a (heterogeneous) environment of existing resources. As a modeling framework, an abstract object model along with the logic-based object specification language `TROLL` has been presented. We have demonstrated our approach by modelling a very simple CIM application example.

The approach should include a set-oriented query facility, which has not been done yet. We think that techniques developed in the context of OODBMS [1, 5] can be used. Further work must also been done on translating `TROLL` specifications into data models and applications (and back) as well as on re-specifying existing services.

Acknowledgements

We are grateful to Cristina Sernadas who developed the basic features of `TROLL` with us. For many fruitful discussions on the language we are grateful to all other members of IS-CORE, especially to Hans-Dieter Ehrich, José Fiadeiro, and Amílcar Sernadas.

References

[1] Atkinson, M. et al.: The Object-Oriented Database System Manifesto. In: Kim, W.; Nicolas, J.-M.; Nishio, S. (eds.): *Proc. DOOD'89*, Kyoto, 1989. pp. 40–57.

[2] Brodie, M. L.; Ceri, S.: On Intelligent and Cooperative Information Systems: A Workshop Summary. *Int. Journal of Intelligent and Coorperative Information Systems*, Vol. 1, No. 2, 1992.

[3] Brodie, M. L.: The Promise of Distributed Computing and the Challenges of Legacy Systems. In: Gray, P. M. D.; Lucas, R. J. (eds.): *Proc. BNCOD 10*. LNCS 618, Springer-Verlag, Berlin, 1992, pp. 1–28.

[4] Buchmann, A. P.: Modeling Heterogeneous Systems as an Active Object Space. In: Dearle, A.; Shaw, G. M.; Zdonik, S. B. (eds.): *Proc. 4th Int. Workshop on Persistent Object Systems*, 1990. Morgan Kaufmann, 1991, pp. 279–290.

[5] Dittrich, K. R.; Dayal, U.; Buchmann, A. P. (eds.): *On Object-Oriented Databases*. Springer, Berlin, 1991.

[6] Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*. Springer, Berlin, 1985.

[7] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: Saake, G.; Sernadas, A. (eds.): *Information Systems – Correctness and Reusability*. TU Braunschweig, Informatik Bericht 91-03, 1991, pp. 1–24.

[8] Ehrich, H.-D.; Saake, G.; Sernadas, A.: Concepts of Object-Orientation. In: *Proc. 2nd IS/KI Workshop Modellierung", Ulm (D)*. Springer IFB 303, 1992, pp. 1–19.

[9] Fiadeiro, J.; Sernadas, C.; Maibaum, T.; Saake, G.: Proof-Theoretic Semantics of Object-Oriented Specification Constructs. In: Meersman, R.; Kent, W.; Khosla, S. (eds.): *Object-Oriented Databases: Analysis, Design and Construction*. North-Holland, 1991, pp. 243–284.

[10] Greenspan, S.; Borgida, A. T.; Mylopoulos, J.: A Requirements Modelling Language and its Logic. In: Brodie, M. L.; Mylopoulos, J. (eds.): *On Knowledge Base Management Systems*. Springer, Berlin, 1986, pp. 471–502.

[11] Garcia-Molina, H.; Salem, K.: SAGAS. In: *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 249–259, 1987.

[12] Hartmann, T.; Jungclaus, R.: Abstract Description of Distributed Object Systems. In: Tokoro, M.; Nierstrasz, O.; Wegner, P. (eds.): *Proc.*

ECOOP'91 Workshop on Object-Based Concurrent Computing. Springer, LNCS 612, 1992, pp. 227–244.

[13] Hartmann, T.; Jungclaus, R.; Saake, G.: Aggregation in a Behavior Oriented Object Model. In: Lehrmann Madsen, O. (ed.): *Proc. ECOOP'92.* Springer, LNCS 615, Berlin, 1992, pp. 57–77.

[14] Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, 1985.

[15] Jungclaus, R.; Saake, G.; Hartmann, T.; Sernadas, C.: Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[16] Jungclaus, R.; Saake, G.; Sernadas, C.: Formal Specification of Object Systems. In: Abramsky, S.; Maibaum, T. (eds.): *Proc. TAPSOFT'91, Brighton.* Springer, Berlin, LNCS 494, 1991, pp. 60–82.

[17] Kaul, M.; Drosten, K.; Neuhold, E. J.: ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views. In: *Proc. 6th Int. Conf. on Data Engineering*, 1990, pp. 2–10.

[18] Kappel, G.; Schrefl, M.: Using an Object-Oriented Diagram Technique for the Design of Information Systems. In: Sol, H. G.; Hee, K. M. van (eds.): *Dynamic Modelling of Information Systems.* North-Holland, Amsterdam, 1991, pp. 121–164.

[19] Lipeck, U. W.: Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science*, Vol. 76, 1990, pp. 115–142.

[20] Mylopoulos, J.; Borgida, A.; Jarke, M.; Koubarakis, M.: Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, Vol. 8, No. 4, 1992, pp. 325–362.

[21] Monarchi, D. E.; Puhr, G. I.: A Research Topology for Object-Oriented Analysis and Design. *Communications of the ACM*, Vol. 35, No. 9, 1992, pp. 35–47.

[22] Saake, G: Conceptual Modeling of Database Applications. In: Karagiannis, D. (ed.): *Proc. 1st IS/KI Workshop, Ulm (Germany), 1990.* Springer, Berlin, LNCS 474, 1991, pp. 213–232.

[23] Saltor, F.; Castellanos, M.; Garcia-Solaco, M.: Suitability of Data Models as Canonical Models for Federated Databases. *ACM SIGMOD Record*, Vol. 20, No. 4, 1991, pp. 44–48.

[24] Sernadas, A.; Fiadeiro, J.; Sernadas, C.; Ehrich, H.-D.: The Basic Building Blocks of Information Systems. In: Falkenberg, E.; Lindgreen, P. (eds.): *Information System Concepts: An In-Depth Analysis.* North-Holland, 1989, pp. 225–246.

[25] Sheth, A. P.: Semantic Issues in Multidatabase Systems (Preface to Special Issue). *ACM SIGMOD Record*, Vol. 20, No. 4, 1991, pp. 5–9.

[26] Saake, G.; Jungclaus, R.: Specification of Database Applications in the TROLL-Language. In: Harper, D.; Norrie, M. (eds.): *Proc. Int. Workshop Specification of Database Systems, Glasgow, 1991.* Springer, London, 1992, pp. 228–245.

[27] Saake, G.; Jungclaus, R.: Views and Formal Implementation in a Three-Level Schema Architecture for Dynamic Objects. In: Gray, P.M.D.; Lucas, R.J. (eds.): *Proc. BNCOD 10.* Springer, LNCS 618, Berlin, 1992, pp. 78–95.

[28] Saake, G.; Jungclaus, R.; Ehrich, H.-D.: Object-Oriented Specification and Stepwise Refinement. In: de Meer, J.; Heymer, V.; Roth, R. (eds.): *Proc. Open Distributed Processing, Berlin, 1991 (IFIP Transactions C, Vol. 1).* North-Holland, 1992, pp. 99–121.

[29] Sheth, A.; Larson, J.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, Vol. 22, No. 3, 1990, pp. 183–236.

[30] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Stocker, P.M.; Kent, W. (eds.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, 1987, pp. 107–116.

[31] Schek, H.-J.; Weikum, G.; Schaad, W.: From Extensibility to Interoperability between Database Systems and Application Systems. In: Kambayashi, Y.; Rusinkiewicz, M.; Sheth, A. (eds.): *Proc. 1st Int. Workshop IMS'91*, Kyoto, 1991. IEEE Computer Society Press, 1991.

[32] Wächter, H.; Reuter, A.: The ConTract Model. In: Elmagarmid, A. K (ed.): *Database Transaction Models for Advanced Applications.* Morgan-Kaufmann, Palo Alto, 1992, pp. 219–263.