# Relationships between Dynamic Objects*

Ralf Jungclaus
Thorsten Hartmann
Gunter Saake

*Abt. Datenbanken, TU Braunschweig*
*P.O. Box 3329, D-W 3300 Braunschweig, FRG*
*E-mail:* {jungclau,hartmann,saake}@idb.cs.tu-bs.de

**Abstract.** In this paper, we describe the specification of communication relationships between objects in a framework for object-oriented conceptual modeling of information systems. In our approach, objects have an observable state and can evolve by the occurrence of events. The possible behavior of objects over time is modeled by a set of admissible object life cycles. A high-level approach to specify systems of interacting objects should provide a means to specify communications between objects explicitly. We introduce language features for the specification of communications and present an approach to transform such relationships into communication channels in order to make implementation easier.

## 1 Introduction

Object-oriented approaches towards conceptual modeling of dynamic object systems have become very popular. We are of the opinion, however, that a pure object-oriented approach (as promoted by the Object-Oriented Programming community [1]) is not suitable in the conceptual modeling phase. In this stage of development, the system to be developed is described on a very abstract level as seen from the application domain. Object-oriented programming approaches support an *operational* view instead of a more *descriptive* view as required in conceptual modeling.

Today, most object-oriented approaches towards domain analysis or UoD-modeling clearly handle structural and dynamic properties *separately* (see e.g. [2, 3]). On the one hand, the components of the system to be modeled are represented in an object model, on the other hand functional or data flow models are used to describe the behavior of the system. At least in the data flow part, aspects local to objects are mixed up with global issues like communication.

Our approach tries to integrate the description of local properties (be they static or dynamic) into object descriptions. Thus, object descriptions are the units of design. Objects have a local state and a behavior over time which is the sequence of local state transitions (the *life cycle* of an object). Each state transition can be regarded as an operation on the object's state. State transitions are called *events* in our approach since

*In: H. Kangassalo; H. Jaakkola; K. Hori; T. Kitahashi (eds.): Infomation Modelling and Knowledge Bases IV: Concepts, Methods and Systems (Proc. 2nd European-Japanese Seminar 1992, Ellivuori (SF)), IOS Press, Amsterdam, 1993, pp. 425 - 438

we do not distinguish between suffered or initiated operations. An object description specifies the attributes that characterize the observable state of an object and the possible life cycles an object can go through. Objects can interact by synchronization of events and the exchange of data values during synchronization. A system is modeled as a collection of interacting objects.

Our approach is formally founded. We are of the opinion that the development of safer systems must start already in the early stages as pointed out in [4]. A conceptual specification should be regarded as the formal basis of system development.

We think that *relationships* are an essential feature of object-oriented conceptual modeling approaches. Relationships between dynamic objects are, however, not as simple as relationships in e.g. the Entity-Relationship model – here, we must be able to specify interactions between objects and dependencies between objects. Other authors have pointed out the need to have relationships in high-level object-oriented modeling [2], but those relationships describe only static relationships between object references. We can identify the following reasons for supporting explicit relationships in conceptual object-oriented modeling:

- Global properties should be separated from local properties of objects. This becomes important when we are in the progress of embedding components into a system context.
- We are able to model communication between components and dependencies between components *explicitly*. It is unnatural to bury information about relationships into the related objects.
- Explicit relationships help to keep context-dependent interrelationships out of object descriptions. This supports *reusability* of objects (which now do only contain local descriptions) and make the local assessment or verification of object descriptions (or component descriptions) possible. This clearly helps in reducing the costs to verify or validate system specifications.
- A conceptual model also describes the environment of the system to be implemented. Thus, relationships between objects in the system and objects of the environment can turn out to be descriptions of the functionality of interfaces to the system. Interfaces are introduced by the implementation of the system and are thus not a part of the initial conceptual model of the organization.

Relationships between dynamic objects are essentially communications since all state changes depend on events. It is straightforward to transform relationships into objects that represent communication channels between related instances. This way, we end up in having a model that consists only of objects.

In this paper, we will describe how relationships are specified in a language for object-oriented conceptual modeling of dynamic systems and how such high-level specification constructs are transformed into low-level channel objects. Before focusing on this topic let us first give a brief introduction into our approach to object-oriented conceptual modeling.

# 2 An Approach to Object-Oriented Conceptual Modeling

Object-oriented conceptual modeling in our view should be integrating the advantages of work on algebraic specification of data types [5, 6] and databases [7, 8], process specification [9, 10], the specification of reactive systems [11], conceptual modeling [12, 13, 14, 15, 16, 17] and knowledge representation [18, 19, 20]. Our approach bases on the one presented in [21]; further introductions into more recent developments are [22, 23].

In a nutshell, object-oriented conceptual modeling approaches organize system specifications as a collection of discrete object descriptions that incorporate both the description of static local aspects and dynamic local aspects of entities (or objects). Object descriptions are thus encapsulated units of structure and behavior description. An object instance has an internal state that can be observed and changed exclusively through an object *interface*. In contrast to object-oriented programming languages that emphasize a functional manipulation interface (i.e. *methods*), object-oriented databases put emphasis on the observable structure of objects (through *attributes*). We propose to support both views in an equal manner, i.e. the structural properties of objects may be observed through attributes and the behavior of objects may be manipulated through *events* which are abstractions of state changing operations or methods.

The encapsulation of all local aspects in object descriptions implies that object descriptions are the *units of design*. Following this perspective, we may model the system *and* its environment in a uniform way. We achieve in having clean interfaces between components that are part of the environment and components that are to be computerized later on. This approach results in having higher levels of modularity and abstraction in the early phases of system design.

An object description usually is regarded as a description of possible instances of the same kind which is similar to the notion of *type* in semantic data modeling. In our terminology, an object description is called a *template*. In object-oriented programming, the notion of type is closely related to (and sometimes even mixed up with) the notion of *class*. In our view, a class defines a *collection* of instances of the same type. A *class type* then is a template along with an *identification space*, i.e. the set of possible identifiers for instances. The class type describes the potential set of possible instances, whereas a class can be regarded as being a container for the actual time-varying collection of existing instances.

Objects can be composed from other objects (*aggregation*). Aggregation of objects imposes a *part-of* relation on a collection of object-descriptions. This kind of inheritance is known from semantic data models where it is used to model objects that appear in several roles in an application.

Object descriptions may also be embedded in a *specialization* hierarchy. Usually, specialization implies reuse of specification code and allows to treat instances both as instances of the base class and the specialization class. A related concept is *generalization* that allows to treat conceptually different instances uniformly as instances of the generalization class.

Besides the structuring mechanisms mentioned above, a means to describe the *interaction* of object instances is needed to specify system dynamics. For conceptual modeling, we have to abstract from implementation-related details that arise from using message-passing and process communication. Communication is modeled conceptually by *calling* or *identifying* events of interrelated objects.

Last, but not least system descriptions should be organized in several levels of modularity. That is, we may want to compose a system not only from objects but from subsystems. Therefore, we need to have a means to *relate* objects from several subsystems. Thus, a framework for object-oriented conceptual modeling should support *relationships* between objects.

In the semantic domain, we regard objects to be observable processes that communicate through synchronous interactions. Processes are sets of sequences of event occurrences that start with a birth event (creation) and that either end with a death event (destruction) or go on infinitely. Each element of a process is called an *object life cycle*. A finite prefix of an object life cycle is called a *history*. A process describes the dynamics of an object instance. An observable state of objects in our approach is a set of attribute-value pairs. The current observable state of an instance is described by a mapping that associates an observable state with each possible history of the object. A detailed description of the semantic domain can be found in [24, 25, 26, 27].

# 3 Language Features for OO Conceptual Modeling

In this section let us introduce the language TROLL [25, 26] with some examples. A complete introduction to the language can be found in [28].

In a *class specification*, we specify two things:

- The *identification space*, i.e. the sort of identifiers for instances and
- the *template*, i.e. a generic description of instances.

The identification space is defined similar to primary key in data models: we define components of a tuple data type. The sort defined in this way represents the possible identifiers for instances. An identification space is notated as `|C|` for a class `C`. For details see [28].

The template consists of a *signature declaration* and a *specification*. In the signature declaration, we introduce the attribute and event symbols. These symbols make up the interface of an instance, i.e. we may read the values of the attributes and may invoke or call events from outside the instance. Attributes can be characterized as being **derived**, i.e. their value is computed from the values of other attributes.

In the event section, we must declare at least one birth event and may declare death events. Other events denote state changes and may have parameters. Parameters have a name and a type and may be marked with the keywords **in** and **out** which give hints about the data flow during communication: **in**-parameters are set by the environment, **out**-parameters are set by the object and delivered to the environment when the event occurs.

The specification consists of sections for **derivation**, **constraints**, **valuation**, and **behavior**.

In the **derivation**-section, we specify how the values of derived attributes depend on the values of other attributes. In the **constraints**-section, we may specify the possible values of attributes or the possible evolution of attribute values over time. Constraints are formulas of future tense temporal logic [29, 30]. Thus, constraints allow us to declaratively specify the behavior or evolution of instances.

In the **valuation**-section, we specify how the values of attributes are altered by event occurrences. They can be regarded as being restricted post-conditions for event occurrences similar to expressions of dynamic logic [31].

In the **behavior**-section, we specify the object life cycles. Due to the variety of behaviors, we may use different formalisms:

- **permissions** are enabling conditions for events – only if no permission is violated in the current state, an event may occur. Permissions may also refer to previous states of the instance.

- **obligations** are requirements to be fulfilled by every admissible life cycle of the instance.
- **commitments** specify reactions that should occur if certain conditions are fulfilled in the current state.
- **patterns** are explicit specifications of parts of object life cycles similar to process specifications in the language CSP [9].

Let us illustrate class specifications with the specification of an object class `ATM` which describes automatic teller machines (ATMs) that can be found at almost each corner today and allow you to withdraw money by using a cash card:

```
class ATM
  identification
    No:nat;
  template
    attributes
      CashOnHand:money;
      derived Dispensed:bool;
    events
      birth set_up; death remove;
      ready; cancel;
      read_card(in C:|CashCard|);
      check_card_w_bank(in Acct:nat:, in PIN:nat);
      card_accepted; bad_PIN_msg; bad_account_msg;
      issue_TA(in Acct:nat,in Amount:money);
      TA_failed_msg; eject_card;
      dispense_cash(in Amount:money);
    constraints
      initially CashOnHand = 10000;
    derivation
      Dispensed = (CashOnHand < 100);
    valuation
      variables m:money;
      [dispense_cash(m)]CashOnHand = CashOnHand - m;
    behavior
      permissions
        variables n:nat; m:money; C:|CashCard|;
        { not Dispensed } read_card(C);
        { m <= CashOnHand } issue_TA(n,m);
      patterns
      variables n,p:nat; m:money;
        process CARD = read_card -> (exists n,p:nat)(check_card_w_bank(n,p))
        end process
        process TA(n) = (exists m:money)(issue_transaction(n,m) ->
                        case
                        dispense_cash(m);
                        TA_failed_msg
                        esac)
        end process;
        (* this is the behavior of the ATM *)
```

```
    CARD -> case
            bad_account_msg -> eject ; (* or *)
            bad_PIN_msg -> eject ; (* or *)
            card_accepted -> TA -> eject
            esac
end class ATM;
```

The behavior of an ATM for example is quite complex. Our language permits (among other possible ways to specify dynamic behavior) to use a process language to describe the behavior. In the ATM example, we declare the subprocesses `CARD` and `TA(n)`. `CARD` says that a `read_card` event occurrence is followed by an occurrence of `check_card_w_bank(n,p)`. The values of the event parameters of `check_card_w_bank` are set by the environment, thus we have to quantify them existentially. In the `TA(n)` subprocess, an occurrence of the `issue_transaction(n,m)` event is followed by either one of `dispense_cash(m)` or `TA_failed_msg`. Please note that the parameter `n` of the process `TA` has to be regarded as an input parameter. The overall behavior of an ATM during one service session is modeled by the process starting with the `CARD` subprocess – it is followed by one of the subprocesses in the **case** selection statement.

The second example is that of a single object that models a bank. Note that we only give a fragment of the specification – space does not permit to model the bank in more detail here. We list only the relevant events that represent important state transitions. We do not model the behavior of the bank concerning these events in this example. The interested reader can find a detailed specification of the bank example in [28].

```
object Bank
  template
    events
      birth establish; death close_down;
      ....
      verify_card(in Acct:nat,in PIN:nat,in ATM:|ATM|);
      no_such_account(out ATM:|ATM|);
      bad_PIN(out ATM:|ATM|);
      card_OK(out ATM:|ATM|);
      process_withdrawal(in Acct:nat,in Amount:money,in ATM:|ATM|);
      TA_failed(out ATM:|ATM|); TA_OK(out ATM:|ATM|);
      ....
end object Bank;
```

Last but not least let us introduce the object class `ATMCustomer` which is modeled to be a *role class* of the object class `Person`. This means that instances of `Person` can be seen as instances of the object class `ATMCustomer` (see also [32, 33]). An object can start playing a role several times in its life and it can play several different roles at the same time. Special events are used for starting and ending to play a role.

```
class ATMCustomer
  role of Person:
  template
    attributes
      HasCards:set(|CashCard|);
    events
      birth bc_atm_customer; death cease;
      active insert_card(in C:|CashCard|,in into:|ATM|);
```

```
    active enter_PIN(in PIN:nat,in into:|ATM|);
    card_accepted(out by:|ATM|);
    active enter_amount(in M:money,in into:|ATM|);
    active enter_cancel(in into:|ATM|);
    cash_dispensed(out by:|ATM|);
    card_ejected(out by:|ATM|;
    active take_card(in from:|ATM|);
    active take_cash(in from:|ATM|);
    active request_card(in Acct:nat);
    card_assigned(in C:CashCard);
    active give_card_back(in C:CashCard);
  valuation
    variables C:|CashCard|;
    [card_assigned(C)]HasCards = insert(C,HasCards);
    { in(C,HasCards)} => [give_card_back(C)]HasCards = remove(C,HasCards);
  behavior
    permissions
      variables C,C1:|CashCard|; atm:|ATM|;
      { in(C,HasCards)} insert_card(C,atm);
      { sometime after(insert_card(C1,atm))
        since last after(insert_card(C,atm))}
          enter_PIN(p,atm),enter_cancel(atm);
      { sometime after(card_accepted(atm))
        since last after(insert_card(C,atm))}
          enter_amount(m,atm),enter_cancel(atm);
      { sometime after(cash_dispensed(atm))
        since last after(insert_card(C,atm))}
          take_cash(atm);
      { sometime after(card_ejected(atm))
        since last after(insert_card(C,atm))}
          take_card(atm);
    commitments
      { after(card_read(atm))} ==> enter_PIN(p,atm) or enter_cancel(atm);
      { after(card_accepted(atm))} ==>
        enter_amount(m,atm) or enter_cancel(atm);
      { after(cash_dispensed(atm))} ==> take_cash(atm);
      { after(card_ejected(atm))} ==> take_card(atm);
end class ATMCustomer;
```

In this example, we specify the behavior of an ATMCustomer by permissions and commitments. This way, we may describe fairly unrestricted behavior patterns. In the **permissions**-section, we state preconditions for event occurrences. These preconditions may also refer to states or event occurrences in the past – we may use a past tense temporal logic dialect to specify permissions. Permissions state that something bad cannot happen in the life of an ATMCustomer instance. In the **commitments**-section we specify that once the preconditions hold in the current state of an object, it has to fulfill the goals as soon as possible in an active way.

Please note that some events are declared to be **active**. These are the events that can occur on the initiative of an instance of ATMCustomer. This way we are able to model the unpredictable behavior of objects in the environment of the system to be implemented.

# 4 Language Features for the Specification of Relationships

In this section, we now want to describe the language features in TROLL that support the putting of separately defined objects or object classes into a global context, i.e. how systems of interacting objects can be build by relating components.

Recall that object descriptions and class definitions are the units of design. In the problem domains we want to model we encounter organizational structures that do not correspond to some complex object – remote and separate objects are connected.

In TROLL, we allow for the specification of *global constraints* and *global communications*. Global constraints make the specification of inter-object dependencies possible. Global constraints enable us to specify how the dynamic behavior of a system of objects is composed from the local behavior of objects and the interaction between objects. For dynamic information systems, this is the most crucial part. We will not say anything about global constraints in this paper but will elaborate on global communications.

As a first example let us describe how an ATM (which is assumed to operate remotely) communicates with the `Bank`.

```
relationship RemoteTransaction between Bank, ATM
  interaction
    variables atm:|ATM|; n,p:nat; m:money;
    /* Card checking business */
    ATM(atm).check_card_w_bank(n,p) >> Bank.verify_card(n,p,atm);
    Bank.no_such_account(atm) >> ATM(atm).bad_account_msg;
    Bank.bad_PIN(atm) >> ATM(atm).bad_PIN_msg;
    Bank.card_OK(atm) >> ATM(atm).card_accepted;
    /* bank transaction business */
    ATM(atm).issue_TA(n,m) >> Bank.process_withdrawal(n,m,atm);
    Bank.TA_failed(atm) >> ATM(atm).TA_failed_msg;
    Bank.TA_OK(atm,m) >> ATM(atm).dispense_cash(m);
end relationship RemoteTransaction;
```

In the example, we state *calling relationships* between events of the `Bank` instance and an `ATM` instance. Event calling means asymmetric synchronous communication. By specifying synchronous communication we abstract from implementation-related issues concerning communication protocols.

The calling relation `e1 >> e2` between two events `e1` and `e2` states that whenever `e1` occurs, then `e2` must occur, too. `e2`, however, may occur without `e1` being obliged to occur also. Thus, we are able to express a *causality* in communications.

In our example, we specified point-to-point communication. By the selector `ATM(atm).` (with the variable `atm` bound to an identifier for instances of the class `ATM`) we refer to exactly one instance of the class `ATM`.

Variables in selectors and events are used to describe the data flow during communication. In the first calling clause,

```
ATM(atm).check_card_w_bank(n,p) >> Bank.verify_card(n,p,atm);
```

the variables `atm`, `n` and `p` are bound to values in the calling event `ATM(atm).check_card_w_bank(n,p)`. The called event `Bank.verify_card(n,p,atm)` then uses these bindings.

Let us now specify the communication between a user of an ATM and an ATM. In this specification, we specify precedence relationships between callings and thus describe a *script*. These precedence relationships are specified using *temporal preconditions* for events. They refer to the history of the communicating instances and guard the interaction: A specified communication can only occur if the precondition holds in the current state.

```
relationship UseATM between ATMCustomer,ATM
  interaction
    variables C:|ATMCustomer|; atm:|ATM|; CC:|CashCard|; p:nat; m:money;
    Customer(C).insert_card(CC,atm) >> ATM(atm).read_card(CC);
    { after(Customer(C).insert_card(CC,atm))} ==>
      Customer(C).enter_PIN(p,atm) >>
        ATM(atm).check_card_w_bank(CashCard(CC).ForAcct,p);
    Customer(C).enter_cancel(atm) >> ATM(atm).cancel;
    { after(Customer(C).enter_PIN(p,atm))} ==>
      ATM(atm).card_OK >> Customer(C).card_accepted(atm);
    { after(Customer(C).insert_card(CC,atm))} ==>
      Customer(C).enter_amount(m,atm) >>
        ATM(atm).issue_TA(CashCard(CC).ForAcct,m,atm);
    { after(Customer(C).enter_amount(m,atm)) } ==>
      ATM(atm).dispense_cash(m) >> Customer(C).cash_dispensed;
    Customer(C).take_cash(atm) >> ATM(atm).eject_card;
    { after(Customer(C).insert_card(CC,atm))} ==>
      ATM(atm).eject_card >> Customer(C).take_card(atm);
    Customer(C).take_card(atm) >> ATM(atm).ready;
end relationship;
```

Take e.g. the following clause:

```
    { after(Customer(C).insert_card(CC,atm))} ==>
      Customer(C).enter_PIN(p,atm) >>
        ATM(atm).check_card_w_bank(CashCard(CC).ForAcct,p);
```

The communication can only take place if the guarding condition is true, here if the temporal term **after**`(Customer(C).insert_card(CC,atm))` is true.

Let us now describe how relationships can be represented as communication channel objects.

# 5 Relationships as Communication Channels

In order to implement relationships, we have to transform them into more primitive structures. Basically, the approach is to transform a relationship into a *channel object* that is shared between the communicating instances. Each communication clause is transformed into a communication event of the channel object.

The goal is to achieve in having an equivalent representation of the system of communicating objects with just one concept (the object concept) instead of two (objects and relationships). Relationships are very convenient for modeling, but for implementation they must be transformed into simpler operational structures. Using the channel approach described below, actors and communications are described uniformly. It is quite straightforward to look at a structure of base objects and channel objects as an abstract description of a distributed object system [34].

A few words about object inclusion are in order here. Object inclusion is the basic mechanism to put objects together. Object inclusion means that an object is included in another object such that no local property of the included object is violated. That means that the state of the included object cannot be altered directly from outside that object – this has to be done by interaction. It also means that an included object can only behave in the specified way. It may, however, be restricted in that it is not able to go through each admissible life cycle anymore.

Let us now describe the channel object that is the result of transforming the relationship `RemoteTransaction`.

```
object class RemoteTransaction
  identification
    ATM:|ATM|;
  template
    events
      birth set_up; death close;
      check_card(n,p);
      no_such_account;
      bad_PIN;
      card_OK;
      issue_TA(n,m);
      TA_failed;
      TA_OK(m);
end object class RemoteTransaction;
```

The resulting channel object is very simple since in the relationship `RemoteTransaction` we do not impose any precedence relationships on the communications.

To establish instances of the object class `RemoteTransaction` as communication channel between the `Bank` instance and the `ATM` instances we must modify the templates of the object `Bank` and the object class `ATM`. We must specify that a specific instance of the object class `RemoteTransaction` is shared by an ATM and the bank as a channel object. Furthermore, we must specify the interactions between events in the base objects and the events in the channel object that represent the communications. For the `Bank` object, this is specified in the following way:

```
object Bank
  template
    including R in RemoteTransaction;
    ...
    events
      birth establish; death close_down;
      ....
      verify_card(in Acct:nat,in PIN:nat,in ATM:|ATM|);
      no_such_account(out ATM:|ATM|);
      bad_PIN(out ATM:|ATM|);card_OK(out ATM:|ATM|);
      process_withdrawal(in Acct:nat,in Amount:money,in ATM:|ATM|);
      TA_failed(out ATM:|ATM|); TA_OK(out ATM:|ATM|);
      ....
    interaction
      variables atm:|ATM|,n,p:nat,m:money;
      RemoteTransaction(atm).check_card(n,p) >> verify_card(n,p,atm);
      no_such_account(atm) >> RemoteTransaction(atm).no_such_account;
      bad_PIN(atm) >> RemoteTransaction(atm).bad_PIN;
      ...
```

```
end object Bank;
```

That is, the occurrence of the check_card event in an instance of the channel object class RemoteTransaction triggers the occurrence of the verify_card event in the Bank object. Please note that we include *all* possible instances of the class RemoteTransaction into the Bank object – an instance of RemoteTransaction represents a communication channel to exactly one instance of the object class ATM.

Now let us see how the specification of the object class ATM is modified:

```
object class ATM
  identification
    No:nat;
  template
    including instance R in RemoteTransaction
        where R.ATM = SELF.No as Channel;
    ...
    interaction
      variables n,p:nat; m:money;
      set_up >> Channel.open;
      check_card_w_bank(n,p) >> Channel.check_card;
      Channel.no_such_account >> bad_account_msg;
      ...
end object class ATM;
```

Here, we include only one instance of the object class RemoteTransaction: The one of which the key value is the same as the key value of the current instance (the identifiers are still different, though). This instance represents the communication channel between one instance of the object class ATM and the object Bank. In our language, we may also rename the instance – here, we give the name Channel to it.

Now we are able to see how the channel works. It is opened when the corresponding ATM is set up – this is specified by the clause

```
set_up >> Channel.open;
```

Whenever the event check_card_w_bank instantiated with the suitable parameters occurs in an instance of the class ATM, then the event check_card in the corresponding channel object is called, which itself calls the event verify_card instantiated with the corresponding values for the event parameters. This is illustrated in Figure 1.

The life cycle of a channel object consists of occurrences of the communication events. All these occurrences are synchronized with the corresponding event occurrences in the Bank object and in the associated instance of the class ATM.

# 6 Conclusions

In this paper, we showed how composite systems may be specified using the TROLL-language. We presented the basic concepts of our approach and gave a brief introduction into the language TROLL. We put emphasis on the specification of relationships between components and showed how communication relationships can be specified using the **relationship** concept of TROLL. We suggested to transform relationship specifications into channel objects in order to make the specification more uniform since we only have to deal with one concept after the transformation. This makes the implementation of specifications easier.

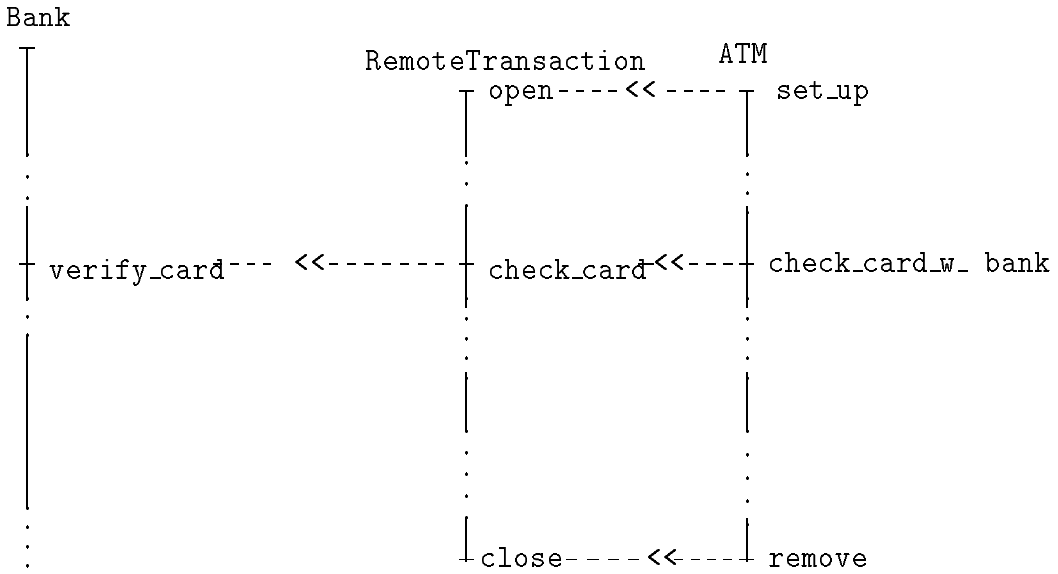


Figure 1: Communication through a channel object

The relationship-concept of the high-level language TROLL is regarded as being very useful. It helps to separate global properties from local properties of objects and thus supports further structuring of conceptual system specifications and reuse of encapsulated components.

For the implementation of conceptual models, however, such high-level constructs should be represented in terms of simpler concepts. We think that a structure consisting of base objects and channel objects is easier to implement since it directly represents the structure of a distributed system.


# Acknowledgements

This work is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-1 and Sa 465/1-2 and by CEC under ESPRIT-II Basic Research Action Working Group No. 3023 IS-CORE (Information Systems – COrrectness and REusability).

Thanks to Cristina Sernadas for developing the basic concepts of TROLL with us. The comments of the anonymous referees are greatly acknowledged.


# References


[1] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOP Messenger*, 1(1):7–87, 1990.

[2] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[3] G. Booch. *Object-Oriented Design*. Benjamin/Cummings, Menlo Park, CA, 1990.

[4] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23:8–24, September 1990.

[5] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer-Verlag, Berlin, 1985.

[6] H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, Stuttgart, 1989.

[7] H.-D. Ehrich. Key Extensions of Abstract Data Types, Final Algebras, and Database Semantics. In D. Pitt et al., editors, *Proc. Workshop on Category Theory and Computer Programming*, pages 412–433. Springer Verlag, Berlin, 1986.

[8] H.-D. Ehrich, K. Drosten, and M. Gogolla. Towards an Algebraic Semantics for Database Specification. In *Proc. 2nd IFIP WG 2.6 Working Conf. on Knowledge and Data (DS-2)*, pages 119–135, Albufeira (Portugal), 1986. North-Holland, 1988.

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

[10] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, 1989.

[11] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Vol. 1: Specification.* Springer-Verlag, New York, 1992.

[12] P.P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[13] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Interactive Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2):185–207, 1980.

[14] M. Brodie, J. Mylopoulos, and J. W. Schmidt. *On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages.* Springer-Verlag, Berlin, 1984.

[15] A. Borgida. Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software*, 2(1):63–73, 1985.

[16] U. Hohenstein and M. Gogolla. A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In C. Batini, editor, *Proc. 7th Int. Conf. on the Entity-Relationship Approach*, pages 129–148, Rome, 1988. North-Holland, Amsterdam, 1988.

[17] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual Modelling of Database Applications Using an Extended ER Model. *Data & Knowledge Engineering*, 1992. North-Holland, *To appear.*

[18] M. L. Brodie and J. Mylopoulos, editors. *On Knowledge Management Systems.* Springer-Verlag, Berlin, 1986.

[19] J. W. Schmidt and C. Thanos, editors. *Foundations of Knowledge Base Management.* Springer-Verlag, Berlin, 1989.

[20] J. Mylopoulos and M. Brodie, editors. *Readings in Artificial Intelligence & Databases.* Morgan Kaufmann Publ. San Mateo, 1989.

[21] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P. Hammerslay, editor, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116, Brighton (GB), 1987. Morgan-Kaufmann, Palo Alto, 1987.

[22] A. Sernadas and H.-D. Ehrich. What Is an Object, After All? In R. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. IFIP WG 2.6 Working Conference DS-4*, pages 39–70, Windermere (UK), 1990. North-Holland, Amsterdam, 1991.

[23] G. Saake, R. Jungclaus, and H.-D. Ehrich. Object-Oriented Specification and Stepwise Refinement. In J. de Meer, V. Heymer, and R. Roth, editors, *Proc. Int. IFIP Workshop on Open Distributed Processing*, IFIP Transactions C: Communication Systems, Vol. 1, pages 99–121, Berlin, 1991. North-Holland, Amsterdam, 1992.

[24] H.-D. Ehrich and A. Sernadas. Fundamental Object Concepts and Constructions. In G. Saake and A. Sernadas, editors, *Information Systems – Correctness and Reusability. (Workshop IS-CORE '91, ESPRIT BRA WG 3023, Selected Papers)*, London, 1991. TU Braunschweig, Informatik-Bericht 91-03, 1991.

[25] R. Jungclaus, G. Saake, and C. Sernadas. Formal Specification of Object Systems. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91 Vol. 2*, pages 60–82, Brighton, 1991. LNCS 494, Springer-Verlag, Berlin, 1991.

[26] R. Jungclaus, G. Saake, and T. Hartmann. Language Features for Object-Oriented Conceptual Modeling. In T.J. Teory, editor, *Proc. 10th Int. Conf. on the ER-approach*, pages 309–324, San Mateo, 1991.

[27] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a Behavior Oriented Object Model. In *Proc. European Conference on Object-Oriented Programming (ECOOP'92)*, pages 57–77, Utrecht (NL), 1992. LNCS 615, Springer-Verlag, Berlin, 1992.

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

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

[30] G. Saake. Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, 6(1):47–74, 1991. North-Holland.

[31] J. Fiadeiro and A. Sernadas. Logics of Modal Terms for System Specification. *Journal of Logic and Computation*, 1(2):187–227, 1990.

[32] R. J. Wieringa. *Algebraic Foundations for Dynamic Conceptual Models*. PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[33] R. Wieringa and W. de Jonge. The Identification of Objects and Roles – Object Identifiers Revisited. Technical Report IR–267, Vrije Universiteit, Amsterdam, 1991.

[34] T. Hartmann and R. Jungclaus. Abstract Description of Distributed Object Systems. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proc. ECOOP'91 Workshop on Object Based Concurrent Computing*, pages 227–244, Geneva (CH), 1991. LNCS 612, Springer-Verlag, Berlin, 1992.