Abstract Description of Distributed Object Systems*†

Thorsten Hartmann
Ralf Jungclaus

Abt. Datenbanken, TU Braunschweig
P.O. Box 3329, D-W-3300 Braunschweig, FRG
E-mail: {jungclau,hartmann}@infbs.uucp

Abstract

In this paper we propose an abstract object-oriented model to describe distributed systems in an actor-oriented way. We introduce *Basic Object Structures* (BOS) with the basic concepts *base objects* to represent entities and *channels* as representations of communications between base objects. Objects are regarded as processes that can be observed. Based on these concepts we show how BOS can be uniformly described using an abstract language, the TROLL-kernel.

1 Introduction

It is rather straightforward to regard a (distributed) system as a collection of dynamic autonomous objects that communicate. This approach dates back to [Hew77] and has initiated different approaches towards a computational model for objects. The *ACTOR* model of computation [Agh86] promotes asynchronous communication between actors, whereas in POOL-T [Ame87] objects are considered as processes that communicate synchronously. Other approaches are ABCL/1 [YSTH87], that models objects as active entities with a single thread of control which may communicate synchronously and asynchronously, or *Abacus* [Nie90, NP90b], that uses a computational model of communicating agents based on CCS [Mil89]. The behavioral description of systems of communicating agents has been a vivid field of research since the presentation of CSP [Hoa85] and CCS.

In this paper we propose a more abstract view of dynamic object systems. We regard objects as units of concurrency encapsulating a single process without internal concurrency (this is known as the *trace model*). The state of an object is described by a number of (structured) attributes that take values from predefined codomains. The attribute values are determined by the history of the object — in this way advantages of process models are combined with data modeling capabilities. Mathematical models of objects in our sense are described in [SE90, ES91].

*This work is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-1 and Sa 465/1-2.

†In: Object-Based Concurrent Computing, Proc. ECOOP'91 Workshop, Geneva, 1991, Tokoro, M. and Nierstrasz, O. and Wegner, P. (eds.), Springer Verlag, Berlin , LNCS 612, pages 227–244

Based on this view we develop *Basic Object Structures* (BOS) to serve as an abstract view of loosely coupled distributed systems. The only concepts supported by BOS are simple (non-composite) objects and communication objects representing communication channels between simple objects. The goal is to provide a suitable interface to implement BOS on a network of workstations.

BOS are the basis for high level abstractions like specialization, generalization, temporary roles, and complex (composite) objects. These abstractions are supported by a language called TROLL particularly suited to the abstract description of interactive information systems [JSH91, JHSS91, SJE91]. Declarative TROLL specifications will be decomposed into equivalent specifications of BOS described using a TROLL-sublanguage, the TROLL-kernel. In this sense BOS can be seen as a step towards implementation of distributed information systems. They provide a link between the conceptual model of the application area and the implemented system.

The advantage of this approach is basically the uniform language framework. In the process of describing a system we start with a consistent model of the application using language features of TROLL. Then we will go on with transformations that do not violate the constraints given by this model. Each step can (at least in principle) be verified against the former step. The result of this procedure is a model of the system in terms of Basic Object Structures described with language features of the TROLL-kernel.

In information systems, we typically have a huge number of *persistent* objects of which just a few are *active* (i.e. providing services and are given CPU resources) at one point in time [SE90]. Thus, the implementation of BOS will be supported by object stores managing the currently passive objects, object managers scheduling the active objects and several communication primitives. This viewpoint also provides a way to weaken the artificial boundary between active and passive objects.

The concept of object used as a basis for TROLL and the TROLL-kernel has been developed in [SSE87, SFSE89b, SE90] and [SFSE89a] which takes a temporal perspective of abstract object types and concurrency. This approach is accompanied by work on a categorical semantics [ES89, EGS90, ES91]. Based on this formal concept, work has been done towards a logical framework of structured theories over a suitable logical calculus [FSMS90, FM91].

The paper is organized as follows: In section 2 we describe the general concept of object in a formal representation. Then we introduce our notion of Basic Object Structures using a simple example of communicating objects. In the main section we describe the language features of the TROLL-kernel used to model BOS providing a modeling of the simple communication example between a server and a possible client. In the last section we give some concluding remarks and state our future plans.

2 Formalization of Object-Oriented Concepts

In this section, we describe how a model, in the mathematical sense, representing a system as a collection of interacting objects is characterized. In our approach, emphasis is put on the representation of the behavior of objects through time and of their evolvement depending on their behavior resulting in a complete representation of structural *and* behavioral aspects.

Let us start with the characteristics of a mathematical model of single isolated objects, called an *object model* in the sequel. An object is in a certain state at any moment in time. Each state transition corresponds to an occurrence of an *event*, which is an abstraction of a method. A set of events may occur simultaneously, which is said to be a snapshot. A sequence of events starting from creating the object (*birth event*) and either ending with deleting the object (*death event*) or going on "forever" is called a *life cycle*. The possible behavior of an object thus is modeled by a set of admissible life cycles which is called a *process*.

Certain aspects of the object's (internal) state may be *observed* using *attributes* that represent *visible* properties of an object. The values of the attributes (an *observation*) depend on the object's state which is determined by the object's history, i.e. it depends on the sequence of state transitions occurred so far. The history is a finite prefix of an admissible life cycle.

Let us now sketch the formalization of these characteristics. Let X be a set of events.

Definition 2.1 *A subset $s \in 2^X$ is called a* SNAPSHOT *over X.* ■

An object event alphabet is composed of *birth, death* and *update* events. We require the set of birth events to include at least one event.

Definition 2.2 *A set of events X is said to be an* OBJECT EVENT ALPHABET *iff $X = B_X \cup D_X \cup U_X$ such that $B_X \neq \emptyset$ and B_X, D_X, U_X are pairwise disjoint.* ■

Snapshots can be classified according to birth or death events they contain. Birth snapshots contain at least one birth event and no death events, death snapshots contain at least one death event and no birth events.

Definition 2.3 *The set of* BIRTH SNAPSHOTS *$\mathcal{B}_X$ is defined as $\mathcal{B}_X = \{s \in 2^X \mid \exists b \in B_X : b \in s, \forall e \in s : e \notin D_X\}$. The set of* DEATH SNAPSHOTS *$\mathcal{D}_X$ is defined as $\mathcal{D}_X = \{s \in 2^X \mid \exists d \in D_X : d \in s, \forall e \in s : e \notin B_X\}$.* ■

For the definition of life cycles we need the notion of finite and infinite life cycles. Infinite life cycles describe objects that will never be destroyed.

Definition 2.4 *The set of* FINITE LIFE CYCLES *over X is denoted as $\mathcal{L}_X^* = \{s_1 \dots s_n \mid n \in I\!N finite,\ s_1 \in \mathcal{B}_X, s_n \in \mathcal{D}_X, \forall i \in I\!N, 2 \leq i \leq n-1 : s_i \notin \mathcal{B}_X \cup \mathcal{D}_X\}$. The set of* INFINITE LIFE CYCLES *over X is denoted as $\mathcal{L}_X^\omega = \{s_1 s_n \dots \mid s_1 \in \mathcal{B}_X, \forall i \in I\!N, i \geq 2 : s_i \notin \mathcal{B}_X \cup \mathcal{D}_X\}$.* ■

The set of life cycles over X is then defined as the union of the empty, finite and infinite life cycles. The empty life cycle ϵ describes the fact that an object has not been created yet.

Definition 2.5 *The set of* LIFE CYCLES *over X is defined as $\mathcal{L}_X = \{\epsilon\} \cup \mathcal{L}_X^* \cup \mathcal{L}_X^\omega$.* ■

Now, we are able to define the behavioral component of an object model, determining it's possible *states*.

Definition 2.6 *A* PROCESS *P is a pair (X, Λ) where X is a set of events and $\Lambda \subseteq \mathcal{L}_X$ is the set of permitted life cycles over X such that $\epsilon \in \Lambda$.* ■

We now formalize how a process determines the observations. Let A be a set of attributes. Associated with each attribute $a \in A$ is a data type $\mathtt{type}(a)$ that determines the range of the attribute. Let $\Pi(\Lambda)$ be the set of all finite prefixes of life cycles in Λ and $\mathtt{carrier}(\mathtt{type}(a))$ be the carrier set for the data type $\mathtt{type}(a)$.

Definition 2.7 *An* OBSERVATION STRUCTURE *V over a process P is a pair (A, α) where A is a set of attributes and α is an observation mapping $\alpha \colon \Pi(\Lambda) \rightarrow obs(A) \subseteq \{(a, d) \mid a \in A \wedge d \in \mathtt{carrier}(\mathtt{type}(a))\}$. A given prefix $\pi \in \Pi(\lambda)$ is called (possible)* STATE *of an object.* ■

Note that by definition we allow for non-deterministic observations.

An object model is defined by putting together a process and an observation structure over that process yielding a complete description of possible behavior and evolution over time.

Definition 2.8 *An* OBJECT MODEL *$ob = (P, V)$ is composed of a process P and an observation structure V over P.* ■

Thus, an object is an *observable process*. As a notational convention, the sequence of observations during the execution of a life cycle $\lambda \in \Lambda$ is denoted by $\omega \in \Omega$, the set of all possible sequences of observations.

Recall that a system is represented by a collection of interacting objects. Thus, we need a mechanism to relate object models. This mechanism is an *object morphism* [EGS90, ESS90]. A special case of object morphism is the *object embedding morphism*. It describes an embedding of an object ob_2 as an encapsulated entity into another object ob_1.

This implies that the set of life cycles of ob_2 must be preserved and ob_1-events must employ ob_2-events to alter ob_2's state. The latter requirement is captured by the concept of *calling*. If an event e_1 calls an event e_2, then whenever e_1 occurs, e_2 occurs simultaneously.

Let us now define an object embedding morphism between objects ob_1 and ob_2. Assume, ob_2 should be embedded into object ob_1. Firstly the events of ob_2 are included in the set of events of ob_1. For the life cycle of ob_1 we state that if we constrain a life cycle to the events of the embedded (or part) object, we have to obtain a valid life cycle of the embedded object. Secondly the attributes of ob_2 are included in the set of attributes of ob_1. For observations of ob_1 projected to the attributes of the part object, we must obtain the same observation as for applying the observation mapping of part object ob_2 to a life cycle of ob_1 restricted to the events of ob_2.

Formally, this is expressed in the following definition:

Definition 2.9 *Let $ob_1 = (P_1, V_1)$ and $ob_2 = (P_2, V_2)$ be object models. Then ob_2 is embedded into ob_1, $ob_2 \hookrightarrow ob_1$, iff*

$$X_2 \subseteq X_1$$

$$\forall \lambda_1 \in \Lambda_1 \exists \lambda_2 \in \Lambda_2 \colon \lambda_1 \downarrow\downarrow X_2 = \lambda_2$$

$$A_2 \subseteq A_1$$

$$\forall \tau_1 \in \Pi(\Lambda_1): \alpha_1(\tau_1) \downarrow A_2 = \alpha_2(\tau_1 \downarrow\downarrow X_2)$$

■

$\lambda_1 \downarrow\downarrow X_2$ denotes life cycle restriction. All events that are not in X_2 are eliminated from the snapshots in λ_1 and only the sequence of non-empty snapshots is taken into account. A single arrow on observations like in $\alpha_1 \downarrow A_2$ only takes into account those attributes that are in A_2.

Usually, objects are classified. For this purpose, we introduce the notion of a class type.

Definition 2.10 *A* Class Type $ct = (I, ob)$ *consists of a set* I *of object identifiers and a (prototype) object model* ob*.* ■

The set I is the carrier set of an arbitrary abstract data type. Following this definition, each object of a class has the same object model as structure definition. This definition can be generalized to heterogeneous classes as proposed in [SSE87] and [JSS91].

Object classes in our setting are sets of objects of a certain type fixed by the corresponding class type. Thus, with each object class we associate one corresponding class type. A class type describes possible extensions of a class definition in terms of object sets identified by identifier values $i \in I$. Therefore, possible class populations are subsets of $\{\langle i, ob\rangle \mid i \in ct.I \wedge ob = ct.ob\}$. Usually, the term *object class* is connected with the current state of a class variable, i.e. a concrete current object population with current states for included objects.

Definition 2.11 *A state of an* Object Class $oc = (ct, O)$ *consists of a class type* ct *and a set of instances* O*. Instances* $\langle i, ob, \pi\rangle \in O$ *are described by an object identifier* $i \in I$*, an object model* $ob = ct.ob$ *and a current state* $\pi \in \Pi(ob.\Lambda)$*. The identifier* i *identifies instances in* O *uniquely.* ■

In this way, an object class definition is extensional. Note that we use the "dot notation" to refer to components of structures.

Taking object models as objects and object morphisms as arrows, a category of objects is established. Models representing the system to be developed can be constructed using category-theoretic operations. For details see [EGS90].

3 Basic Object Structures

Before we describe BOS consider a tiny example. Suppose we want to model a stopwatch. With the `start`-event, the stopwatch is created and starts to count the time units. The passing of one time unit is indicated by an occurrence of the `tick`-event. The `stop`-event models the pushing of the stop-button. A `deliver`-event parameterized by a natural number delivers the count of time units and is only permitted to occur after the `stop`-event has occurred. Our kind of stopwatch can only be used once, the `reset`-event is the death event of a stopwatch. The two observable properties of stopwatches are the `time` being the count of time units that passed since the start and a flag `stopped` that indicates whether the `stop`-event has occurred so far. Each admissible life cycle starts with a `start`-event, continues with an arbitrary number of occurrences of the `tick`-event before a `stop`-event. There may or may not be a `deliver`-event after the `stop` before the `reset`-event. Using an informal notation, resembling the specification of an object model (see definition), this may be written as follows:

```
stopwatch =  (  {  time:nat, stopped:bool },
                {  start, tick, stop, deliver(nat), reset },
                {  start → tick → stop → reset,
                   start → tick → ... → stop → reset,
                   start → tick → stop → deliver(1) → reset,
                   start → tick → tick → stop → deliver(2) → reset,
                   ...
                   start → tick → ... → stop → deliver(n) → reset,
                   ... },
               ...
             )
```

Note that we omitted the specification of the observation mapping and that the process description is incomplete showing only the intuitive behavior of the proposed stopwatch.

The basic mechanism to relate objects is *object inclusion*. Briefly, it is an embedding morphism between object models which allows to *include* an object into another without violating any of its properties (i.e. only local events may alter its state and the set of possible life cycles must be preserved, see above). Only inside such a composite object we may have *communication*. The basic mechanism for communication is asymmetric synchronous communication which is called *event calling*. If an event e_1 calls an event e_2 ($e_1 >> e_2$), then e_2 must occur synchronously whenever e_1 occurs, but e_2 may occur alone. *Event sharing*, denoted by $e_1 == e_2$, means that e_1 and e_2 always occur synchronously (i.e. they are identified).

Now let us turn to *Basic Object Structures* (BOS). In BOS, only two kinds of objects are known. Firstly, there are simply-structured *base objects* representing some kind of entity as stopwatches, customers, users and so on. The second kind of objects are called *channels* and represent the "communication channels" between base objects that communicate, for example communication buffers or message queues.

Basically, channels include events that model communications between base objects. A channel is *shared* by base objects which are specified to synchronize. This is modeled by including the channel into the communicating base objects. Thus, the base objects *know* about the properties of the channel objects. This way they can force events of the channel object to occur using the concept of calling or sharing. On the other hand also the channel objects may induce the occurrence of events of the base objects in the same way. Conceptually, the channels are considered to be *parts* of the base objects.

Consider an example. Suppose we have a type `STOPWATCH` that describes stopwatch instances as described above. As name space we provide an isomorphic copy of the set of natural numbers called |`STOPWATCH`|, where the operation `STOPWATCH: nat` → |`STOPWATCH`| takes a natural number to generate a unique surrogate. In general, we may identify in-

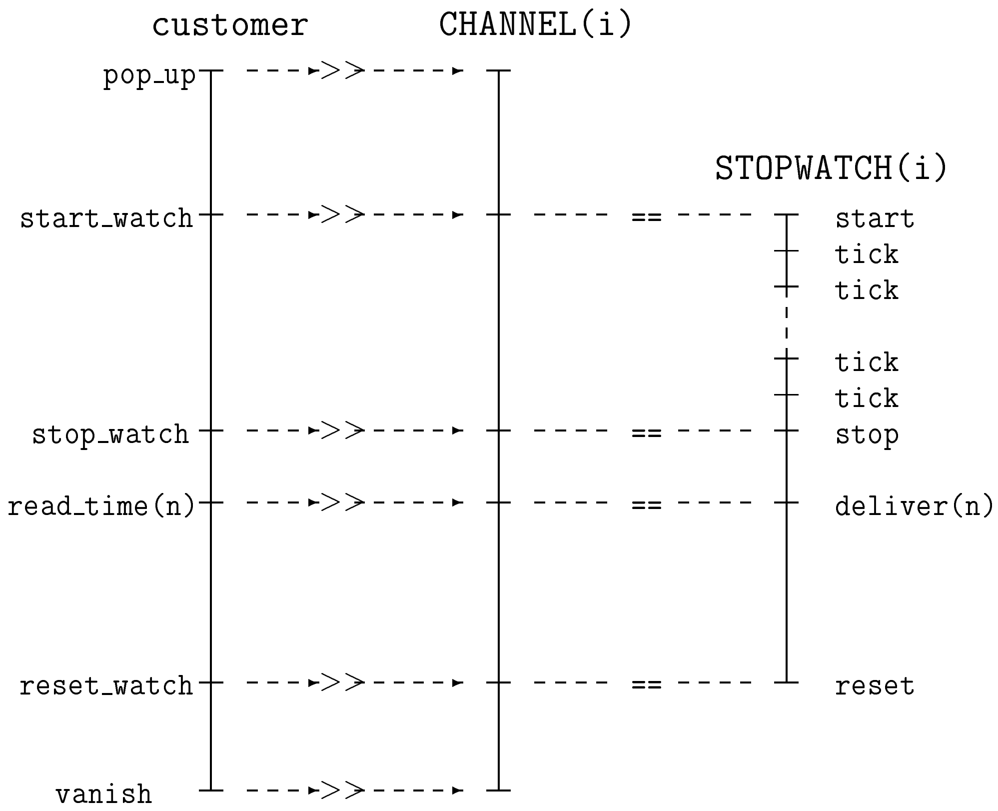

Figure 1: Communication of base objects through a channel

stances by giving the classname and a natural number like e.g. `STOPWATCH`(15). Stopwatches, in this example the server objects, are used by a customer, the client. The `customer` is created and destroyed independently from stopwatches with the events `pop_up` and `vanish`, respectively. Additionally, there are events `start_watch`, `stop_watch`, as well as `read_time(nat)` and `reset_watch` that correspond to the events of stopwatches.

Figure 1 shows how an instance of the class `CHANNEL` models the communication between `customer` and an instance of `STOPWATCH`. The `CHANNEL` object serves as a buffer between instances of clients and servers.

In this example events of the `customer` are immediately reflected in events of the `CHANNEL` and therefore also in events of the `STOPWATCH`, i.e. we have synchronous communication. Asynchronous communication may be modeled using channels with attributes, through which they are able to "remember" previous states. This kind of channel objects may delay service requests using queues for example.

In the following section, we introduce language features of the TROLL-kernel to describe *Basic Object Structures*.

4 Language Features to Describe BOS

The TROLL-kernel includes language features for abstract description of BOS. It is upward-compatible with the language TROLL that supports the description of information systems on higher levels of abstraction.

A specification of the structure of (prototype) objects is called a *template*. A template is structured as follows:

template [`template name`]
  data types *data types to be imported;*
  attributes *attribute names and types;*
  events *event names and parameter types;*
  valuation *effects of events on observations;*
  permissions *enabling conditions for events;*
  obligations *completeness requirements for life cycles;*
[**end template** `template name`]

Template names are optional and may be used to refer to separately defined templates, i.e. code reuse in class type definitions. Let us now describe the subsections of a template specification. In the data type section, the data types to be imported for the underlying data universe are indicated. The data types may either be chosen from a set of predefined types (like 'nat', 'string' or 'bool') or from a set of user-defined types specified using a language for abstract data type specification. Note that we also may import name spaces which are defined by an abstract data type (implicitly with the specification of a class type). Such name spaces are denoted by |`OCT`| for an object class type `OCT`. The type constructors (**list**, **set**, **record**, **bag**), known from abstract data type specifications, may be applied to construct complex structured data types. For example the constructor **set** applied to the predefined data type `nat`: **set**(`nat`) yields a complex structured data type representing sets of integers with operations like:

`insert`: `nat` $\times$ **set**(`nat`) $\rightarrow$ **set**(`nat`)
`delete`: `nat` $\times$ **set**(`nat`) $\rightarrow$ **set**(`nat`)
`empty`: **set**(`nat`) $\rightarrow$ `bool`
...

In the **attribute**-section, the attribute names and the codomains are specified. In the **event**-section, event names and parameters are introduced. Event parameters allow for data to be exchanged during communication and for defining the effects of events on attribute values. The symbols introduced in the data type section, the attribute section and the event section make up the *local signature* of an object specification.

As an example, consider the local signature of the `STOPWATCH`-template:

template `stopwatch`
  data types `nat, bool`;
  attributes
    `time:nat`;
    `stopped:bool`;

```
events
    birth start;
    tick;
    stop;
    deliver(nat);
    death reset;
...
end template stopwatch
```

The signature defines the access interface for an object, that is, the external visible properties in terms of observations and possible event names. Another terminology for the signature of TROLL objects would be visible state and possible operations. At this point nothing is said about the *correct* use, i.e. the underlying semantics of the object. Signatures (in this stage) can only be used for type checking but not for reasoning about object behavior and correct use [FM91, NP90a]. On a higher level of abstraction TROLL allows to define special interfaces to objects, thus restricting the visible properties of an object to a proper subset [JHSS91, SJ91].

In the **valuation**-section of a template, rules that specify the effects of event occurrences on the attribute values are presented. The rules are implicitly universally quantified and invariant. A valuation rule has one of the following forms:

```
[event] attribute = data_term;
```

or:

```
[event] attribute_1 = data_term_1,
        attribute_2 = data_term_2,
        ...
        attribute_n = data_term_n;
```

Such a rule denotes that the attribute `attribute` after the occurrence of the event `event` will have the value of the term `data term`. The data term is evaluated in the state *before* the occurrence of the event and thus corresponds to an assignment in imperative programming languages. The second form is added as 'syntactic sugar', denoting that after the occurrence of event `event` attributes 1,. . .,n will have the values of data terms 1,. . .,n respectively. Another way to look at valuation rules are simple postconditions to event occurrences. An implicit frame rule states that attribute values not being set by a valuation rule remain unchanged after an event has occurred.

As an example, consider the valuation rules for `STOPWATCH`-objects:

```
valuation
    [start] time = 0,
            stopped = false;
    [tick] time = time + 1;
    [stop] stopped = true;
```

After the `start` event of the stopwatch the attribute `time` is initialized to zero and the attribute `stopped` is initialized to **false**. `Tick` events increment the counter attribute `time` and the `stop` event sets the 'flag' attribute `stopped` to true.

In the **permissions**- and the **obligations**-section, the set of admissible life cycles is defined declaratively by giving conditions to be fulfilled by each life cycle. These sections describe the *behavior* component of an object description.

Permissions state preconditions for the occurrence of events. Events are only allowed to occur if the precondition is fulfilled. Roughly, permissions correspond to safety conditions (here: restricting the *next* possible states) [Lam83]. In the TROLL-kernel, preconditions are formulae of a first-order logic, build upon attribute symbols and operations imported in the data type section.

Consider the permissions for `STOPWATCH`es:

> **permissions**
>   **variables** `n:nat`;
>   {**not** `stopped`} `tick`;
>   {**not** `stopped`} `stop`;
>   {`stopped` **and** `n` = `time`} `deliver(n)`;
>   {`stopped`} `reset`;

These rules state, that `tick` and `stop` events are only permitted if the attribute `stopped` is bound to **false**. Note that the `stop` event, since it alters the attribute `stopped`, may only occur once.

In the **obligations**-section, conditions to be fulfilled by complete life cycles are stated. The conditions must be fulfilled before an object can be destroyed. For `STOPWATCH`es, we require the `reset`-event to occur:

> **obligations**
>   `reset`;

In this sense obligations correspond to *liveness requirements*, stating what *must* occur. **Obligations** are thus used to describe necessary events in the live of an object and, in case of passive objects, can be seen as guides for correct use by potential clients. For example, we may have specified a slightly different obligation, stating that a `deliver` event must occur in the life cycle of stopwatches:

> **obligations**
>   ((**exists** `n:nat`) `deliver(n)`) **and** `reset`;

This condition implies, that users of stopwatch objects (clients) must invoke a `deliver` operation before a `reset`, the death event of the stopwatch. Failing to do so is a violation of the *contract* between objects and its clients [Mey88]. This contract can be derived from the life cycle definition of stopwatches. Informally, it may be expressed as: Potential users of stopwatch objects, that issued `start` and `stop` events *must* issue a `deliver` event before destroying the stopwatch object with the `reset` event.

This view of the server object `stopwatch` is purely passive. The other way round, specifying the `deliver` event as an *active* event would state that the stopwatch can force this event to occur, that is, to behave like an active object. Following this line permissions correspond to input offers, whereas obligations correspond to output offers in the terminology of [NP90b, NP90a].

Let us now consider the specification of class types. We want to describe a *set* of objects with similar structure and behavior. Each object in this set or *class* must be uniquely identified. As identification space we use an isomorphic copy of the set of natural numbers. The class type `STOPWATCH` is specified as follows (compiling the former described specification fragments under the keyword **template**):

```
object class STOPWATCH
  identification
    number:nat
  template
    data types nat, bool;
    attributes
      time:nat;
      stopped:bool;
    events
      birth start;
      tick;
      stop;
      deliver(nat);
      death reset;
    valuation
      [start] time = 0,
              stopped = false;
      [tick] time = time + 1;
      [stop] stopped = true;
    permissions
      variables n:nat;
      {not stopped} tick;
      {not stopped} stop;
      {stopped and n = time} deliver(n);
      {stopped} reset;
    obligations
      reset;
end object class STOPWATCH
```

With the definition of a class type an abstract data type (ADT) for object identifiers is generated implicitly. As a convention the name of this ADT is given by the class type name between vertical bars. Note that we may use the identification selector of an instance denoted by `number` in the same way as attribute symbols defined within the object template.

For the sake of completeness, here is the specification for `customer`. In the attribute `uses` we save the identifier of the `STOPWATCH`-instance that is currently used by the customer.

```
object customer
  template
```

```
    data types nat, bool, |STOPWATCH|;
    attributes
      stopped_time:nat;
      stopped:bool;
      uses:|STOPWATCH|;
    events
      birth pop_up;
      start_watch;
      stop_watch;
      read_time(nat);
      reset_watch;
      death vanish;
    valuation
      variables n:nat;
      [pop_up] stopped_time = 0,
               uses = STOPWATCH(1);
      [start_watch] stopped = false;
      [stop_watch] stopped = true;
      [read_time(n)] stopped_time = n;
      [reset_watch] stopped_time = 0,
                    uses = STOPWATCH(uses.number+1);
    permissions
      variables n:nat;
      {stopped_time=0} start_watch;
      {not stopped and stopped_time=0}stop_watch;
      {stopped and stopped_time=n}read_time(n);
end object customer
```

Customers use different stopwatches since stopwatches are destroyed after they have done their work, i.e. a `reset` event occurred. Every time a customer resets the currently used stopwatch and therefore removes it, the attribute `uses` is altered to the next possible stopwatch identifier. The dot notation `uses.number` is another notation for the operation `number(uses)` (yielding integer values) defined implicitly in the abstract data type |STOPWATCH|.

Up to now we have specified two different base objects. The next step is to tie together these base objects using a mediator object, a channel. This `CHANNEL` object handles the communication between the `customer` object and an instance of class `STOPWATCH`. Note that channels are identified by identifiers of `STOPWATCH` instances. We thus have one `CHANNEL` for each instance of `STOPWATCH`.

```
object class CHANNEL
  identification
    with:|STOPWATCH|;
  template
    data types nat;
    events
```

```
        birth set_up;
        startwatch;
        stopwatch;
        readtime(nat);
        resetwatch;
        death close_down;
end object class CHANNEL;
```

These CHANNEL objects only contain events, but this is not a general requirement. Recall that we may specify channels with buffer or queueing functions. Therefore attributes as a form of memory would be needed.

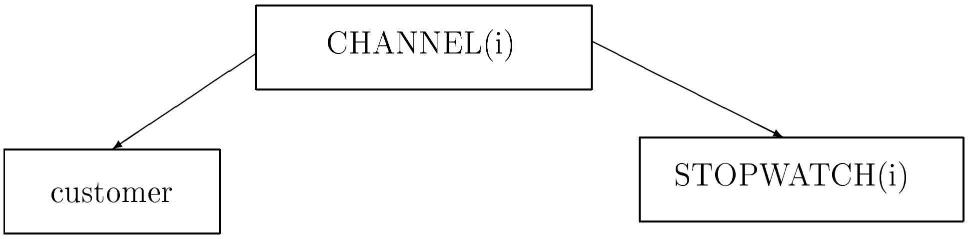

Figure 2: System of communicating objects

Figure 2 shows the relationship between the customer, an instance of STOPWATCH and the CHANNEL.

Now let us consider how the TROLL-kernel can be used to model a system of communicating base objects. The customer must include all instances of CHANNEL after they have been created. Since all communications are initiated by customer, in the **interactions**-section of customer the calls to the communication events must be specified:

```
object customer
  including C in CHANNEL;
  template
    ...
    interactions
      variables n:nat;
      pop_up >> CHANNEL(uses).set_up;
      start_watch >> CHANNEL(uses).startwatch;
      stop_watch >> CHANNEL(uses).stopwatch;
      read_time(n) >> CHANNEL(uses).readtime(n);
      reset_watch >> CHANNEL(uses).resetwatch;
      vanish >> CHANNEL(uses).close_down;
end object customer
```

By definition, the operation CHANNEL, applied to an object identifier, yields the object itself. Recall, that referencing these CHANNEL objects is only possible because they are included in the customer object (see def. 2.9). This way the signature of customer objects also contains the event symbols of `CHANNEL` objects. Since we included a set of (possible) objects, we have to deal with *indexed* symbols. Indexing is denoted using the dot notation.

Similar to `customer` objects, an instance of class `STOPWATCH` must include the corresponding `CHANNEL` instance. Since each `STOPWATCH` instance is associated with exactly one `CHANNEL` instance, a slightly different including clause is used. Here a single instance of class `CHANNEL` is included. The notation `SELF.`**identification** denotes the instance of the data type |`STOPWATCH`| associated with the current instance. The condition `C.with`=`SELF.`**identification** thus binds the identifier `C` to the desired `CHANNEL` instance.

Since there are no name clashes between the signatures of `CHANNEL` and `STOPWATCH` we do not need to rename any components:

```
class STOPWATCH
   identification
      number:nat
   template
      including instance C in CHANNEL
         where C.with=SELF.identification
      ...
      interactions
         variables n:nat;
         start == startwatch;
         stop == stopwatch;
         reset == resetwatch;
         deliver(n) == readtime(n);
end class STOPWATCH
```

In contrast to the `customer` specification, we have sharing of events. This implies that the connection between `CHANNEL`s and `STOPWATCH`es is stronger than between `CHANNEL`s and `customers` because communication is *symmetric*.

Suppose the `STOPWATCH` is modeled as an active object, that is it can force events to occur by own initiative. These events will then be reflected in the `CHANNEL` due to the sharing equations in the interaction section and therefore in other objects possibly connected to it (that have included the `CHANNEL`).

In this case obviously there should be no other objects that use the `STOPWATCH`. We can reason that with this *communication structure* defined with the interactions between the base objects and the `CHANNEL`, the `customer` is not aware of events triggered in the `STOPWATCH`.

Consider for example another client 'connects' to the `CHANNEL` after some `tick` events occurred and forces concurrently to the former client the `stop` event. Note that this situation cannot be distinguished from an active `STOPWATCH` object, since the original client is not aware of concurrent use of the `STOPWATCH` by other clients forcing state changes of the `STOPWATCH`. The result is trouble within the original client: the `STOPWATCH` is stopped, but this state is not reflected in its `stopped` attribute.

Therefore, communication by calling is not sufficient here because events induced to the `STOPWATCH` from somewhere else may not be noticed by the `customer`. We should rewrite the specification of the communication structure between `customer` and `STOPWATCH` to:

```
object customer
    including C in CHANNEL;
    template
        ...
        interactions
            variables n:nat;
            pop_up >> CHANNEL(uses).set_up;
            start_watch == CHANNEL(uses).startwatch;
            stop_watch == CHANNEL(uses).stopwatch;
            read_time(n) == CHANNEL(uses).readtime(n);
            reset_watch == CHANNEL(uses).resetwatch;
            vanish >> CHANNEL(uses).close_down;
end object customer
```

Now state changes of the STOPWATCH that coincide with state changes of the customer (for example the stop and stop_watch events) are reflected symmetrically in both objects. Now a correct use of the STOPWATCH from within the customer is possible, even if there are other concurrent requests to the STOPWATCH operations (even if this is not the desired use of a stopwatch).

5 Conclusions and Future Work

In this paper we have introduced a language for abstract description of Basic Object Structures. We have sketched a model of objects being regarded as processes observed by attributes. Then, we argued that distributed systems can be modeled using base objects and channels. Channels are objects that represent communication relationships between objects. Base objects can be regarded to be the actors in the system. Their behavior described using *permissions* and *obligations* can be used for reasoning about correct use by potential clients.

We think that BOS reflects the structures of distributed systems. Actors (base objects) and channels (communication lines) are described in a uniform way using the TROLL-kernel language. The models of specifications are object models as described in section 2.

Currently, we are starting to investigate how BOS can be implemented on a network of workstations. First ideas are that base objects that are currently active can be mapped to OS-processes. Therefore a more detailed notion of *activity* of objects is needed. Certain events may be specified as being active and can be executed by own initiative.

Channel objects may be mapped to specialized OS-processes performing communication with other processes. Whether this communication takes place between different machines using physical communication lines or between processes on one machine should be transparent, thus allowing objects to migrate. The uniform description of base objects and channel objects using one language framework appears to be the main advantage of this approach.

Another facet of objects in our sense is persistence. Instances may carry a lot of state information (attributes), thus we will need a database system to manage the state information of active *and* passive objects. These are only first ideas towards an implementation of abstract specifications of information systems.

Acknowledgments

We are very grateful to Gunter Saake and Cristina Sernadas who influenced significantly the language design of TROLL. Thanks also to the anonymous referees for providing very interesting ideas to our original workshop submission and not at last to all the participants of the workshop.

References

[Agh86] G.A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA, 1986.

[Ame87] P. America. POOL-T: A Parallel Object-Oriented Language. In M. Tokoro and A. Yonezawa, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, 1987.

[EGS90] H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages (Proc. REX School/Workshop)*, pages 203–228, Noordwijkerhood (NL), 1990. LNCS 489, Springer-Verlag, Berlin, 1991.

[ES89] H.-D. Ehrich and A. Sernadas. Algebraic Implementation of Objects over Objects. In W. deRoever, editor, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (Proc. REX'89)*, pages 239–266, Mood (NL), 1989. LNCS 394, Springer Verlag, Berlin, 1989.

[ES91] H.-D. Ehrich and A. Sernadas. Fundamental Object Concepts and Constructions. In G. Saake and A. Sernadas, editors, *Information Systems — Correctness and Reusability*, pages 97–128. Technical University Braunschweig, Informatik Bericht 91-03, 1991.

[ESS90] H.-D. Ehrich, A. Sernadas, and C. Sernadas. From Data Types to Object Types. *Journal on Information Processing and Cybernetics EIK*, 26(1/2):33–48, 1990.

[FM91] J. Fiadeiro and T.S.E. Maibaum. Towards Object Calculi. In G. Saake and A. Sernadas, editors, *Information Systems — Correctness and Reusability*, pages 129–178. Technical University Braunschweig, Informatik Bericht 91-03, 1991.

[FSMS90] J. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-Theoretic Semantics of Object-Oriented Specification Constructs. In R. Meersman and

W. Kent, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam. *In print.*

[Hew77] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8:323–364, 1977.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[JHSS91] R. Jungclaus, T. Hartmann, G. Saake, and C. Sernadas. Introduction to TROLL – A Language for Object-Oriented Specification of Information Systems. In G. Saake and A. Sernadas, editors, *Information Systems — Correctness and Reusability*, pages 97–128. Technical University Braunschweig, Informatik Bericht 91-03, 1991.

[JSH91] R. Jungclaus, G. Saake, and T. Hartmann. Language Features for Object-Oriented Conceptual Modeling. In T. Teory, editor, *Proc. 10th Int. Conf. on the ER-Approach*, San Mateo (CA), pages 309–324, 1991.

[JSS91] R. Jungclaus, G. Saake, and C. Sernadas. Formal Specification of Object Systems. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91*, pages 60–82, Brighton (UK), 1991. LNCS 494, Springer-Verlag, Berlin.

[Lam83] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, 1989.

[Nie90] O. Nierstrasz. A Guide to Specifying Concurrent Behaviour with Abacus. In D. Tsichritzis, editor, *Object Management*, pages 267–293. Université de Genève, 1990.

[NP90a] O. M. Nierstrasz and M. Papathomas. Towards a Type Theory for Active Objects. In *Object Management*, pages 295–304. Université de Genève, 1990.

[NP90b] O. M. Nierstrasz and M. Papathomas. Viewing Objects as Patterns of Communicating Agents. In N. Meyrowitz, editor, *ECOOP/OOPSLA'90 Proceedings*, Ottawa, 1990. ACM, 1990. also in [NP90a].

[SE90] A. Sernadas and H.-D. Ehrich. What Is an Object, After All? In R. Meersman and W. Kent, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4)*, Windermere (UK), 1990. North-Holland, Amsterdam. *In print.*

[SFSE89a] A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. Abstract Object Types: A Temporal Perspective. In Banieqbal, Baringer, and Pnueli, editors, *Temporal Logic in Specification*. Springer, LNCS 398, 1989.

[SFSE89b] A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. The Basic Building Blocks of Information Systems. In E. Falkenberg and P. Lindgreen, editors, *Information System Concepts: An In-Depth Analysis*, pages 225–246, Namur (B), 1989. North-Holland, Amsterdam, 1989.

[SJ91] G. Saake and R. Jungclaus. Specification of Database Applications in the TROLL Language. In *Proc. Int. Workshop on the Specification of Database Applications*, 1991. To appear.

[SJE91] G. Saake, R. Jungclaus, and H.-D. Ehrich. Object-Oriented Specification and Stepwise Refinement. In *Proc. International IFIP Workshop on Open Distributed Processing, Berlin*. North Holland, 1991. To appear.

[SSE87] 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.

[YSTH87] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1. In M. Tokoro and A. Yonezawa, editors, *Object-Oriented Concurrent Programming*, pages 55–89. The MIT Press, 1987.