In H. Bowman and J. Derrick, editors, Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), Volume 2, IFIP TC6 WG6.1 Intern. Workshop, 21-23 July, Canterbury,Kent, UK, pages 89-104. Chapman & Hall, 1997.

Specifying Distributed Information Systems: Fundamentals of an Object-Oriented Approach Using Distributed Temporal Logic

G. Denker and H.-D. Ehrich
Abteilung Datenbanken, Technische Universität Braunschweig
Postfach 3329, D-38023 Braunschweig, Germany
Voice: +49 +531 391{3103 |3271}, Fax: +49 +531 3913298
{G.Denker|HD.Ehrich}@tu-bs.de

Abstract

We present fundamentals of an approach to object-oriented specification of distributed information systems. We do not assume global time for concurrent object systems. For specifying those systems we propose DTL, a distributed temporal logic. The main contribution is that DTL is capable of specifying complex constraints about the behavior of distributed systems and communication between concurrent objects. For instance, we distinguish different kinds of synchronous communication such as immediate calling vs. deferred calling. The ideas are illustrated by examples given in TROLL, a formal object-oriented specification language. We introduce notations for formally specifying object-based distributed systems. Finally, we briefly explain how DTL is semantically explained in terms of a true concurrent model, i.e., labelled event structures, and which concepts for in-the-large specification are covered by our approach.

Keywords

Object orientation, specification language, distributed system, semantical model, distributed temporal logic, formal description technique, information system, concurrency

1 INTRODUCTION

Most of the work currently done in the field of distributed systems is about networks of processors. Questions concerning hardware, process communication, distributed operating systems, etc. are investigated. Concurrency comes along with distribution. We assume that units of distribution are understood as granules of concurrent behavior. There is a bunch of mathematical models for concurrent processes such as petri nets, event structures, and transition systems together with process logics such as temporal logics or Hennessy-Milner-Logik. Process languages (e.g. CSP or CCS) as well as programming languages (e.g. OCCAM or UNITY) arose from these efforts.

We adopt an object-oriented approach to system specification. Object-oriented concepts are already widely accepted for software programming. Recently object-oriented concepts take the step into early design phases of software systems including analysis and specification. A sound theoretical underpinning is necessary to deal with design issues such as verification, refinement, model checking, etc. Event structures and temporal logics are used as semantical framework in this paper.

We present fundamentals of an approach to specifying distributed information systems. A system is modeled as an interacting collection of sequential objects that operate concurrently. The approach combines ideas from algebraic data type specification, conceptual modeling, database design, behavior modeling, specification of reactive systems, and concurrency theory.

A specification language named TROLL is being developed that is based on these ideas. TROLL is a language especially designed for being used in early phases of information system design. The description of static and dynamic features is integrated in object descriptions. Theses object descriptions are the basic building blocks of system specifications. TROLL supports different abstractions, such as classes, inheritance, static and dynamic specialization, composition, etc. The latest published version is (Jungclaus *et al.* 1995). A new version, still under development, emphasizes distribution and concurrency (Hartel 1996).

In this paper, we elaborate on distribution ideas that are being discussed for inclusion in the next TROLL version. The emphasis here is on theoretical foundations rather than language features. We illustrate our ideas by example using ad-hoc notations.

For specifying concurrent object behavior, we envisage to use DTL, a distributed temporal logic that is an extension of linear temporal logic. DTL is based on n-agent logic (Lodaya *et al.* 1992). Interpretation structures are locally sequential labelled event structures. Objects correspond to sequential substructures. Synchronous and symmetric "handshake" communication is modelled by letting the sequential substructures overlap at shared events. So we have local linear times within objects and synchronization points shared by objects. There is no global time.

The distinguishing feature of this model is that it expresses non-interleaving concurrency while the logic does not explicitly talk about concurrency: we are still close to linear temporal logic with the added possibility of "talking about other objects".

This is achieved by specifying from the local viewpoints of objects in the system, not that of an external observer with a bird's view of the system.

There are some similarities to the approaches of TLA (Abadi *et al*. 1995) a temporal logic of actions to specify concurrent systems, and CTR (Bonner *et al*. 1996) a concurrent transaction logic in which communication is specifiable. In contrast to TLA and to CTR our models are true concurrent models whereas TLA is based on a sequential model and CTR also uses an interleaving model. We use event structures as models. Using them means that we do not assume interleaving semantics. This model implies that we do not have global time nor global space, i.e., shared data. Thus, the way of describing the properties of an open distributed system, e.g. global constraints such as behaviour relations between concurrent objects, is done by communication.

Abadi and Lamport use so called assumption/guarantee specifications which assert that the system provides a guarantee if its environment satisfies an assumption. We only use the communication principle to specify the behavior of concurrent systems. Another difference to their approach is that they are concerned with proving implementation relations between specifications. One of their main issue is to reason about composition of assumption/guarantee specifications. Since our approach is purely based on communication, we will face different problems. It is subject to future work to see how we can deal with implementation problems mentioned in (Abadi *et al*. 1995).

The approach of Bonner and Kifer differs from our in the sense that they specify several concurrent processes which synchronize on a common database. Our approach uses the concept of objects to encapsulate data. Thus, we do not have a central data space on which all processes have to synchronize. Their approach is similar to ours in the respect that they also specify behavior locally to processes, but the programming paradigm of CTR differs from the one that is induced by using distributed temporal logics as the basis for object-oriented languages.

Among other approaches to developing object specification languages with a sound theoretical basis, MAUDE (Meseguer 1993, Clavel *et al*. 1996) also deals with concurrency. It is based on rewriting logic (Meseguer 1992) that is a uniform model of concurrency. GNOME (Sernadas *et al*. 1994) is based on temporal logic, it is most closely related to TROLL although the semantics is not capable of expressing non-interleaving concurrency.

The paper is organized as follows: In the following section we examine the issues that arise in the specification of distributed systems. In particular, we discuss different kinds of synchronous communication in distributed systems by means of examples. Our examples are given in TROLL, a formal object-oriented specification language. In Section 3 we propose a linear time temporal logic (DTL) which can deal with distribution. We show how DTL can be used to give semantics to TROLL specifications. This is done by translating language features to temporal formulas. Thus, TROLL only provides a convenient way of temporal specification. In Section 4 we briefly describe event structure semantics, and in Section 5 we mention the expres-

sive power of our approach since it incorporates different in-the-large concepts for specification. We conclude with a summary and future work.

2 BASIC LANGUAGE CONCEPTS

We demonstrate basic specification concepts by means of a toy example, using an ad-hoc notation in the spirit of TROLL. Some language features for specifying distributed systems have been proposed in Hartel (1996).

Example 1 (BankWorld) We have object classes `Account` and `Bank`. *money* is a predefined data type, *data type* `acct# = [100..999]` is a user-defined data type.

An account has an attribute `balance` and actions `open`, `close`, `credit(`*nat*`)` and `debit(`*nat*`)`. *** and *+* express that the actions may only occur at the beginning or at the end of a life cycle, respectively. The behavior of an object is specified separately from the signature. The effects of actions are specified in *do-od* clauses. Preconditions for actions are specified in *onlyIf* clauses.

```
object class  Account
  attributes
    balance: nat initialized 0;
  actions
    * open;
     credit(money);
     debit(money);
    + close;
   behavior
     credit(m) do balance:=balance+m od;
     debit(m)
       onlyIf balance>=m
       do balance:= balance-m od;
end
```

A bank has a `founder` as attribute of type *string* which will be set when it is created. The founder is a *constant* attribute. Thus, it cannot be changed during the life of a bank object. A bank is a complex object with accounts as components that are identified by account numbers. `transfer(m,from,to,at)` is an action for transferring an amount `m` of money `from` a local account `to` an account at another bank with bank code number `at`. Bank code number is a user-defined data type: *data type* `bankCode# = [1..9]`.

`receive(m,from,at,to)` is an action for receiving an amount `m from` an account `at` bank `to` a local account. The receiver bank acknowledges the receipt of money (`acknowledge(m,from,at,to)`) and the sender bank receives the acknowledgement (`receiveAckn(m,from,to,at)`).

The two banks involved in a transfer operate concurrently. We will come back to distribution issues in the context of `BankWorld`.

```
object class  Bank
  attributes founder:string constant;
    components Accs(acct#):Account;
    actions
      * establish(founder:string);
      transfer(m:money,from:acct#,to:acct#,at:bankCode#);
      receive(m:money,from:acct#,at:bankCode#,to:acct#);
      acknowledge(m:money,from:acct#,at:bankCode#,to:acct#);
      receiveAckn(m:money,from:acct#,to:acct#,at:bankCode#);
      + liquidate;
    behavior
      establish(f) do founder:=f od;
      transfer(m,from,to,at) do Accs(from).debit(m) od;
      receive(m,from,at,to)
        do eventually Accs(to).credit(m),
           next acknowledge(m,from,at,to) od;
    constraints
      transfer(m,from,to,at) and transfer(m,from,to2,at2)
       implies (to=to2 and at=at2);
      transfer(m,from,to,at) implies ( not transfer(m,from,to,at)
         until receiveAckn(m,from,to,at));
      receive(m,from,at,to) implies (Accs(to).credit(m)
         before Accs(x).debit(n));
end
```

The behavior specification says that if a `transfer` happens then the money will be debited in the same step. Thus, the bank communicates with its component. If money is received then it will eventually be put on the appropriate account and the receipt will be acknowledged in the next step.

Furthermore, there are three constraints ensuring correct transaction management: (1) there must not be two different transfers of the same amount from the same account at the same time, and (2) as long as receipt of a transfer is not acknowledged, the same amount may not be transferred from the same account to the same account, and (3) as long as an account is not credited no other account may be debited. The first constraint ensures that no debit is lost, and the second constraint ensures that every transfer is acknowledged. The last constraint ensures that money which has been received will be credited before any other account will be debited. All constraints are implicitly universally quantified and all temporal operators are implicitly bound to bank `self`. In more detail, the second constraint says that the same transfer in `self` is forbidden *until* the acknowledgement has been received so that `self` is sure that bank `at` has received the money. The meaning of the temporal *until* operator ensures that the acknowledgement must eventually happen, so `at` must eventually have received the money.

In the example, object classes are specified as generic structure and behavior templates of objects of the same type. Instances of object classes are called objects.

Objects are concurrently put together to form a distributed system. TROLL offers a way to specify concurrent objects in an *object system*. After declaring all data types and object classes, the possible *objects* of the described world are defined. The *objects* construct provides a set *Id* of object identities. `BankWorld` consists of several concurrently existing `Banks`. These are distinguished by bank code number. Each bank object is a complex object composed of several accounts.

object system `BankWorld`
  data type `acct# ...; bankCode# ...;`
  object class `Account ...` *end*;
  object class `Bank ...` *end*;
  objects `Banks(nr:bankCode#): Bank;`
  behavior `...`
end

Dynamic creation and abortion of objects is treated by * and + actions. So we have a fixed set of "possible" objects while there is a time-varying subset of "actual" objects.

We distinguish between sequential and concurrent objects. Objects are understood as sequential units, even if they are composed. E.g. a bank consisting of several accounts is semantically modelled as a sequential process. Though in each step different accounts may perform actions, the overall behavior of a bank is a sequential process. Furthermore, there is communication between the components of the composed object as well as between the compound object and its components. For example, a bank communicates with an account while transferring money by debiting the account. In our model communication is synchronous. The mentioned example will be reflected in the model by a sequence of actions where in one step both actions (`transfer` and `debit`) happen. Due to the underlying sequential model this kind of communication can be described using linear temporal logics. In a distributed system we need new description techniques for communication.

Different objects are concurrently put together to form a distributed system. For example the banks of `BankWorld` behave concurrently. Going from sequential objects to concurrent systems opens new possibilities. For instance, assume there is a system requirement that can only be fulfilled if different actions from various concurrent objects are executed. Since in our model there is no global observer we can only assure such a requirement by communicating the execution of the corresponding actions among the objects which are incorporated. But such an approach leaves freedom for two parameters: the communication point between the objects and the execution time of the actions they are asked for. To formalize such requirements we need a formalism which does not demands global view. For this purpose we will introduce DTL in the following section. Before doing so we will illustrate communication in distributed systems by means of examples.

For instance, given two banks `Banks(1)` and `Banks(2)` and a transfer action `Banks(1).transfer(m,x,y,Banks(2))`. We want to specify that the transferred money will be received, i.e., `Banks(2).receive(m,x,Banks(1),y)` happens. As-

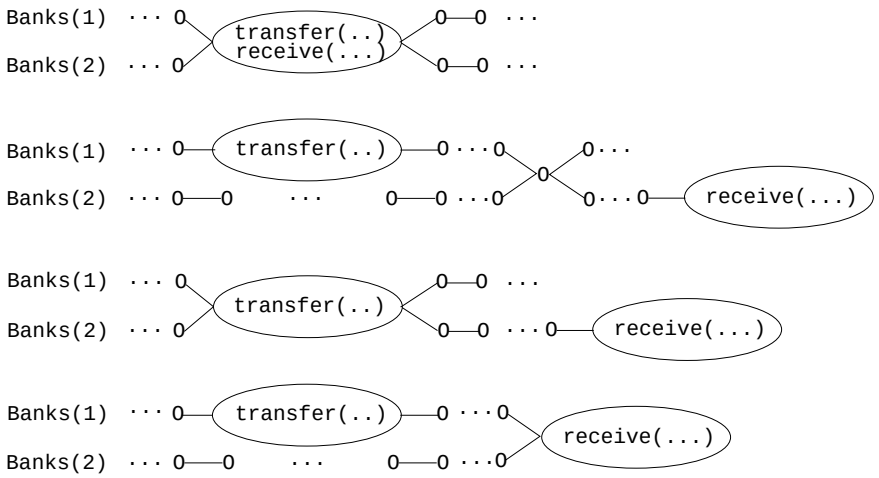

Figure 1 Possible synchronous communication schemas between concurrent objects

suming synchronous communication there are four possibilities to fulfill the requirement that `Banks(1)` is sure that `Banks(2)` received the money:

1. The banks meet and perform the corresponding actions together. This corresponds to synchronous communication in sequential systems. In other words, the transfer action of `Banks(1)` **immediately calls** `Banks(2)` and the latter **immediately executes** the receive action.
2. The meeting point between `Banks(1)` and `Banks(2)` is somewhere in-between the states where the actions `transfer(...)` and `receive(...)`, respectively, are executed. This corresponds to a **deferred calling** from `Banks(1)` to `Banks(2)` and a **deferred execution** of both actions.
3. When `Banks(1)` transfers the money it **immediately calls** `Banks(2)` to get the information that the latter will receive the money sometime in the future (**deferred execution**). This is a special case of 2 where the communication point coincides with the execution of `Banks(1).transfer(...)`. The called action `Banks(2).receive(...)` will be executed later.
4. Another special case of 2 is that the communication point coincides with the execution of the called action, i.e., **deferred calling** and **immediate execution**. Sometime after transferring the money `Banks(1)` contacts `Banks(2)` which is currently receiving the money.

These possibilities are illustrated in Figure 1. Object states are depicted by circles put in sequence by connecting lines, and communication points are depicted by shared states.

Therefore, we have four possible communication structures between concurrent objects. Communication in a sequential object is always immediate. Appropriate language features are still under development, but a possible description is given below.

```
object system BankWorld
  ...
  objects Banks(nr:bankCode#): Bank;
  communication
   Banks(snd)->Banks(rcv):
    transfer(m,from,to,rcv) >> receive(m,from,snd,to)
    transfer(m,from,to,rcv) >F>  eventually receive(m,from,snd,to)
    transfer(m,from,to,rcv) >>  eventually receive(m,from,snd,to)
    transfer(m,from,to,rcv) >F> receive(m,from,snd,to)
   Banks(rcv)->Banks(snd):
    acknowledge(m,from,snd,to) >F> receiveAckn(m,from,to,rcv)
end
```

The first part of the communication refers to the objects involved, i.e., directed synchronous communication between `Banks(snd)` and `Banks(rcv)`.

The second part concerns the actions involved and the kind of communication: immediate communication (`>>`) or deferred communication. For deferred communication there is the choice between `>X>`, i.e., communication in the next state, or `>F>`, i.e., communication in some following state. Moreover, either the called action is immediately executed when the communication takes place or the action is executed in the next (*next*) or some following state (*eventually*).

The last communication axiom says that the acknowledgement given by the receiving bank will sometime be received by the sender bank. The communication will be at the time of receipt. ■

The focus of our work is on mathematical foundations of distributed specifications. Thus, in the following section we will propose a distributed temporal logic (DTL) in which situations like the one given above can be formulated. An appropriate logic must be able to express formulae from the viewpoint of different objects. We show how the given bank example is translated into logical terms using DTL. The general idea is that TROLL only provides a convenient way of specification. Semantically, TROLL specifications are treated by translation to DTL. Thus, one can use TROLL as a language tool for temporal system specification. In the following section we will present how different TROLL features are translated to DTL. Because of lack of space we only illustrate the translation in terms of our example. The full semantics of TROLL is presented in Hartel (1996). The expresiveness of the current language TROLL is much more restrictive than of the underlying logics. This is due to the fact that TROLL is also used in a real-size case study (Krone *et al.* 1996) where intuitive understanding and executability of specification is of great importance.

In our example we focussed on features for specifying distributed, interacting object systems. TROLL offers also in-the-large features like inheritance, generalization, etc. We briefly mention these issues in section 5.

3 DTL SPECIFICATION

Let $\Sigma = (S, \Omega)$ be a data signature where S is a set of sorts and Ω is an $S^* \times S$-indexed set family of operation symbols. A TROLL specification determines structural as well as behavioural aspects of the system. An in-the-small TROLL system specification is formalized as a pair $SysSpec = (\Sigma_I, \Phi)$. $\Sigma_I = (Id, At, Ac)$ is a signature covering all structural aspects, consisting of a set Id of identities, an $Id \times S$-indexed set family $At = \{At_{i,s}\}_{i \in Id, s \in S}$ of attribute symbols, and an Id-indexed set family $Ac = \{Ac_i\}_{i \in Id}$ of action symbols. Without loss of generality we assume that actions do not have parameter. $\Phi = \{\Phi_i\}_{i \in Id}$ is an Id-indexed set family of DTL formulae. A detailed construction of Σ_I from a given TROLL specification is given in Ehrich *et al*. (1995).

Example 2 (BankWorld signature) Let `Banks(1)`, `Banks(2)`, ... be banks. Let `100`, `101`, ... be account numbers. The instance signature of the bank specification is $\Sigma_I = (Id, At, Ac)$, where $Id = \{$`Banks(1)`, `Banks(2)`, ... $\}$, and for `j,k = 1,2`, ..., we have

- $At_{\texttt{Banks(j)},string}$ = {`Banks(j).founder`},
- $At_{\texttt{Banks(j)},money}$ = {`Banks(j).Accs(100).balance`,
  `Banks(j).Accs(101).balance`, ... },
- $Ac_{\texttt{Banks(j)}}$ = {`Banks(j).establish`,
  `Banks(j).transfer(50,100,999,Banks(k))`, ... ,
  `Banks(j).receiveAckn(50,100,999,Banks(k))`, ... ,
  `Banks(j).Accs(100).debit(50)`, ... }.

The attributes and actions of the components of the bank, i.e., its accounts, are imported in the signature of `Banks(j)`. ■

The DTL formulae $\Phi = \{\Phi_i\}_{i \in Id}$ are obtained by translating corresponding language fragments: initialization declarations, action effects and preconditions, synchronization axioms, constraints, etc.

Before we introduce DTL we start with a simpler logic, a basic distributed temporal logic DTL_B, in which immediate calling and execution can be treated easily. We illustrate by example how to translate the sequential part of our example as well as immediate communication to DTL_B formulae. Afterwards we define DTL in which immediate as well as deferred calling and execution, respectively, can be formalized appropriately. But DTL is not more expressive than DTL_B in the sense that every DTL formula can be translated to a DTL_B formula introducing further communication actions (Ehrich *et al*. 1997). DTL is only more convenient for specifying distributed systems. Wrt communication DTL_B corresponds to the actual TROLL version. There communication is expressed by directly defining the actions on which objects synchronise. In DTL it is possible to define deferred communication and execution, among others. DTL_B is defined as follows:

$$\text{DTL}_B ::= \{\text{DTL}_B{}^i\}_{i \in Id},$$

$$\text{DTL}_B{}^i ::= i.H_b^i \mid i.C_b^i,$$

$$H_b^i ::= \text{ATOM} \mid (H_b^i \Rightarrow H_b^i) \mid (H_b^i \; \mathcal{U} \; H_b^i) \mid (H_b^i \; \mathcal{S} \; H_b^i),$$
$$\text{ATOM} ::= \mathit{false} \mid T_\Sigma \; \theta \; T_\Sigma \mid AT_\Sigma \; \theta \; T_\Sigma \mid \triangleright AC \mid \odot AC,$$
$$C_b^i ::= \ldots \mid (\mathcal{C}_i \Rightarrow j.\mathcal{C}_j) \mid \ldots \qquad \text{for } i, j \in Id, \; j \neq i.$$

Basic distributed temporal logic is split into the local *home*-logic $i.H_b^i$ of object i and the communication logic $i.C_b^i$. The local home logic is a propositional linear time temporal logic. An atom is a boolean constant, a pair of data terms, an attribut-data term pair, or a predicate on an action expressing that an action is enabled ($\triangleright a$) or has occurrred ($\odot a$). T_Σ, AT_Σ and AC denote data, attribute and action terms, respectively; θ denotes a comparison operator like $=, \leq, \ldots$. $\mathcal{U}$ and $\mathcal{S}$ are the temporal operators *until* and *since*. The other temporal operators can be derived from these, for instance $\mathsf{X}\ \varphi = \mathit{false}\ \ \mathcal{U}\ \ \varphi$ for *next*, $\mathsf{F}\ \varphi = \varphi \vee \mathit{true}\ \ \mathcal{U}\ \varphi$ for *sometime in the future*, $\varphi\ \mathcal{P}^+\ \psi\ = \neg((\neg\varphi)\ \mathcal{U}^\circ\ \psi)$ for *before*, $\mathsf{G}\ \varphi\ = \neg(\mathsf{F}(\neg\varphi))$ for *always*, $\mathsf{P}\ \varphi\ = \varphi \vee \mathit{true}\ \mathcal{S}\ \varphi$ for *sometime in the past*, etc. (cf. (Ehrich 1996) for details). Let $\mathcal{C}_i$ be communication predicates of object i $\in$ Id, e.g. $\odot a$, $\triangleright b \wedge \odot a$. The communication logic $i.C_b^i$ can express immediate communication. For instance, $i.(\odot a \Rightarrow j.\odot b)$ is a communication formula that means that whenever a happens in object i it immediately synchronizes with object j which will perform b.

The following example shows how some of the relevant specification fragments of the object classes `Bank` and `Account` in example 1 are expressed in DTL_B.

Example 3 (BankWorld axioms) Referring to the signature Σ_I given in example 2, let `Banks(1)` be an instance of class `Bank`. Local formulae of object `Banks(1)` have the form `Banks(1).`(φ), where φ is a home- or a communication formula. We omit the local identity and give in the following only φ to increase readability. From the class specifications in example 1, we obtain the following formulae in $\Phi_{\texttt{Banks(1)}}$.

initialization:
The first axiom translates the *initialized* statement in *object class* `Account`. The second axiom corresponds to the effect of the birth action `establish`:
$\odot$`Banks(1).Accs(acct#).open` $\Rightarrow$ `Banks(1).Accs(acc#).balance` $=$ `0`,
$\odot$`Banks(1).establish(f)` $\Rightarrow$ `Banks(1).founder` $=$ `f`.

constant attributes:
After a founder has been set he/she will remain unchanged for the rest of the bank's life:
`Banks(1).founder` $=$ `f` $\Rightarrow$ (G `Banks(1).founder` $=$ `f` $\mathcal{U}$ $\odot$`Banks(1).liquidate`).

effects of actions:
Specifying effects of actions requires to refer to attribute values in the state before the action was executed:
($\odot$`Banks(1).Accs(from).debit(m)` $\wedge$ Y `Banks(1).Accs(from).balance` $=$ `b`)
$\Rightarrow$ `Banks(1).Accs(from).balance` $=$ `b` $-$ `m`.

preconditions of actions:
$\odot$`Banks(1).Accs(from).debit(m)` $\Rightarrow$ `Banks(1).Accs(from).balance` $>=$ `m`.

constraints:
Constraints are mainly translated one to one from the specification, e.g.
$\odot$`Banks(1).receive(m, from, to, at)`$\Rightarrow$
$\odot$ `Banks(1).Accs(from).credit(m)` $\mathcal{P}^+$ $\odot$`Banks(1).Accs(x).debit(n)`.

immediate communication:
We have to distinguish between immediate and deferred communication. Immediate communication between concurrent objects can easily be formalized with DTL$_B$. Deferred communcation will be illustrated after introducing DTL. The following formula corresponds to the first possibility given in Figure 1.
$\odot$`Banks(1).transfer(m, x, y, z)` $\Rightarrow$ $\odot$`Banks(2).receive(m, x, Banks(1), y)`. ■

We introduce DTL to express deferred kinds of communication. DTL is defined as follows. We concentrate on the propositional fragment:

$$\begin{aligned}
&\text{DTL} ::= \{\text{DTL}^i\}_{i \in Id},\\
&\text{DTL}^i ::= i.H^i,\\
&H^i ::= \text{ATOM} \mid (H^i \Rightarrow H^i) \mid (H^i \; \mathcal{U} \; H^i) \mid (H^i \; \mathcal{S} \; H^i) \mid C^i,\\
&\text{ATOM} ::= \mathit{false} \mid T_\Sigma \; \theta \; T_\Sigma \mid AT_\Sigma \; \theta \; T_\Sigma \mid \triangleright AC \mid \odot AC,\\
&C^i ::= \ldots \mid \text{DTL}^j \mid \ldots \qquad\qquad \text{for } i, j \in Id,\ j \neq i.
\end{aligned}$$

The main difference to DTL$_B$ is the possibility of arbitrarily changing the viewpoint between concurrent objects in a formula. Change of viewpoint implies that communication takes place. For instance, $i.(\odot a \Rightarrow \mathsf{F}\ j.(\odot b))$ means that whenever object i executes a it will sometime in the future meet object j (deferred calling) and object j will execute b at the meeting point (immediate execution). More examples are given below.

DTL as well as DTL$_B$ are propositional, linear time temporal logics. As already mentioned DTL is not more expressible than DTL$_B$, but it is more appropriate for different kinds of communication specification in distributed systems including immediate as well as deferred calling and execution. So far there is no proof system for our logic. This is subject to further research. But nevertheless, since we have a well-defined semantics we can proof semantically properties of the distributed system. Since in a distributed system communication is the technique to exchange information, the interesting properties to be dealt with are those which involve several concurrent objects. Requirements which basically depend on the communication between such objects are of major interest. Our logic provides a way to specify such requirements. Moreover, it is still a linear temporal logic. Properties expected to be analyzable are those which express specific protocols between objects, behavior patterns in distributed systems, behavior that emerges when concurrent objects are composed to systems, etc.

Example 4 (BankWorld axioms (contd)) With the help of DTL the communication axioms given in `BankWorld` are as follows:
immediate and deferred communication:

$$\texttt{Banks(snd)}.[\odot\texttt{Banks(snd).transfer(m, from, to, rcv)} \Rightarrow \odot\texttt{Banks(rcv).receive(m, from, snd, to)}],$$

$$\texttt{Banks(snd)}.[\odot\texttt{Banks(snd).transfer(m, from, to, rcv)} \Rightarrow \mathsf{F}\,\texttt{Banks(rcv)}.[\mathsf{F}\odot\texttt{Banks(rcv).receive(m, from, snd, to)}]],$$

$$\texttt{Banks(snd)}.[\odot\texttt{Banks(snd).transfer(m, from, to, rcv)} \Rightarrow \texttt{Banks(rcv)}.[\mathsf{F}\odot\texttt{Banks(rcv).receive(m, from, snd, to)}]],$$

$$\texttt{Banks(snd)}.[\odot\texttt{Banks(snd).transfer(m, from, to, rcv)} \Rightarrow \mathsf{F}\odot\texttt{Banks(rcv).receive(m, from, snd, to)}],$$

$$\texttt{Banks(rcv)}.[\odot\texttt{Banks(rcv).acknowledge(m, from, snd, to)} \Rightarrow \mathsf{F}\odot\texttt{Banks(snd).receiveAckn(m, from, to, rcv)}].$$ ■

DTL achieves its expressiveness by changing the local view in a temporal formula. Though the logics does not explicitly talk about distribution, distributed systems are formalizable. DTL is by far more expressive than what we exploit to give semantics to the four different communication styles in the TROLL example. To give an idea of its capability we present some formulae involving two distributed objects. We introduce $@u \mathrel{\widehat{=}} u.\mathsf{X}^?\ \mathit{true}$ (*at u, locality*) as an abbreviation. This means that there is currently communication with u. Let $i, u \in Id$ *("I, you")* be object identities:

$i.\,\mathsf{G}(@u \Rightarrow \mathsf{X}\,\varphi)$	whenever I talk to you, you have φ the next day
$i.u.\mathsf{P}\,\varphi$	I hear that φ was once valid for you
$i.(\varphi \Rightarrow \mathsf{X}\,@u)$	if φ is true, then I will talk to you tomorrow
$i.u.\mathsf{X}\,@i$	you tell me that you will contact me tomorrow

DTL is semantically defined with the help of a non-interleaving model. In the following section we briefly present event structure semantics and satisfaction of DTL.

4 EVENT STRUCTURE SEMANTICS

Interpretation structures for DTL are locally sequential labelled event structures (cf. (Winskel *et al.* 1995) for event structures). In (Sassone *et al.* 1993) a classification of concurrency models is given and event structures are considered to be a fair choice when a non-interleaving denotational model is needed. Single objects are modelled by sequential labelled event structures. Models for single objects are sets of sequential life cycles where in each step a set of (component) actions may be performed. This captures the intuition that objects have a state and perform actions changing state.

Formally single objects are modelled by labelled sequential event structures $\mathbf{C} = (E, \lambda, P)$. A sequential event structure $E = (Ev, \rightarrow^*, \#)$ consists of a set of events e, a partial ordering $\rightarrow^*$ (causality) and a symmetric and irreflexive binary relation

(conflict). There is no concurrency in sequential event structures. The labelling $\lambda : Ev \rightarrow P$ maps events to the state logic P, i.e., assertions about attribute values, enabled and occurring actions, etc. Interpretation of DTL will be given in life cycles. A life cycle L of a sequential object is a maximal totally ordered sub-event structure of E, i.e., L is a sequence of events of a sequential object.

For modelling distributed systems, sequential event structures are composed concurrently to form a locally sequential event structure, i.e., an union of sequential event structures overlapping at shared interaction events. Thus, a possible system run corresponds to a distributed life cycle, a concurrent composition of sequential life cycles of system objects.

One of the main differences between sequential objects and concurrent systems is that of concurrency. The concept of concurrency is derived in event structures. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure. Two events $e, e' \in Ev$ are concurrent, $e \ co \ e'$, iff $\neg(e \rightarrow^* e' \vee e' \rightarrow^* e) \wedge \neg e \# e'$. A system $\mathbf{S}$ of objects $i \in Id$ is a triple $\mathbf{S} = (E, \lambda, P)$, where $\mathbf{C}_i = (E_i, \lambda_i, P_i) \subseteq \mathbf{S}$ is a local class of object i in $\mathbf{S}$.

As usual, models of a system specification $SysSpec = (\Sigma_I, \Phi)$ are interpretation structures satisfying the axioms.

Satisfaction of DTL is locally defined. For a given system model $\mathbf{S}$ we define satisfaction of a formulae for a distributed life cycle L at a specific event $e \in L$.

Let $\mathbf{S} = (E, \lambda, P)$ be a system Let L_i be a life cycle of object i and $L = L_1 \cup \cdots \cup L_n$ be a distributed life cycle in $\mathbf{S}$. Let $e \in L$ be an event of the system. Satisfaction $\models_i$ is locally defined for object i. Let $\varphi, \psi \in H^i$, $\gamma \in C^j = D^j$.

$\mathbf{S}, L, e \models i.\varphi$	holds iff $e \in L_i$ and $\mathbf{S}, L, e \models_i \varphi$.
Let $e \in L_i$:	
$\mathbf{S}, L, e \models_i p$	holds iff $\lambda_i(e) \models_{P_i} p$,
$\mathbf{S}, L, e \models_i false$	does not hold,
$\mathbf{S}, L, e \models_i (\varphi \Rightarrow \psi)$	holds iff $\mathbf{S}, L, e \models_i \varphi$ implies $\mathbf{S}, L, e \models_i \psi$,
$\mathbf{S}, L, e \models_i (\varphi \ \mathcal{U} \ \psi)$	holds iff there is a future event $e' \in L_i$, $e \rightarrow^+ e'$, where $\mathbf{S}, L, e' \models_i \psi$ holds, and $\mathbf{S}, L, e'' \models_i \varphi$ holds for all $e'' \in L_i$, $e \rightarrow^+ e'' \rightarrow^+ e'$,
$\mathbf{S}, L, e \models_i (\psi \ \mathcal{S} \ \varphi)$	holds iff there is a past event $e' \in L_i$, $e' \rightarrow^+ e$, where $\mathbf{S}, L, e' \models_i \varphi$ holds, and $\mathbf{S}, L, e'' \models_i \psi$ holds for all $e'' \in L_i$, $e' \rightarrow^+ e'' \rightarrow^+ e$,
$\mathbf{S}, L, e \models_i j.\gamma$	holds iff $e \in L_j$ and $\mathbf{S}, L, e \models_j \gamma$.

At an event e a communication formulae holds iff both objects are sharing e. Thus, whenever the view is changed in a formulae it requires that the involved objects communicate.

With an appropriate morphism concept for locally sequential event structures, the model category of a specification has a final element that may serve as a canonical semantics (Caleiro 1996).

5 IN-THE-LARGE SPECIFICATION

The power of our approach comes to fruition when dealing with in-the-large concepts like inheritance, hiding and composition such as generalization and sequential and concurrent aggregation (Ehrich 1996).

Relevant relationships between models are given by event structure morphisms (Winskel *et al.* 1995). For example, semantic inheritance is modeled by event structure inclusion morphisms expressing that the inheriting object has obtained some of the attributes and actions from the original one. Overriding is covered because event structure morphisms are partial. Hiding, i.e., the relationships between objects and their interfaces, generalized objects and their constituents, complex objects and their components, etc. are also modeled by event structure morphisms.

Event structures and their morphisms form a category **ev**. Sequential event structures and their (total) morphisms form a category **evs**. The main results are the following:

- **ev** is complete and has coproducts.
- **evs** is complete and cocomplete.
- Coproducts in **evs** coincide with those in **ev**.
- These coproducts express generalization, i.e. alternative choice.
- Products in **ev** express concurrent aggregation.
- Products in **evs** express sequential aggregation, i.e. component composition.

The coproduct construction is given in (Winskel *et al.* 1995). For products in **ev**, an elegant construction is given in (Vaandrager 1989). We are not aware that the completeness and cocompleteness results have been published elsewhere. The constructions in **evs** are simple, though. Equalizers in **ev** also have a simple construction. Pullbacks may be utilized for modeling aggregation with sharing, e.g. event sharing for handshake communication. Pushouts in **evs** may be utilized for modeling generalized objects with shared constituents. Coequalizers in **ev** do not exist in general but there are interesting cases where they exist and may be utilized.

6 CONCLUDING REMARKS

The theory outlined here is taking shape but is not complete, it has to be elaborated and refined in several respects. For instance, in-the-large composition aspects have to be worked out. The goal is to establish a high-level algebraic treatment and optimization of concurrent systems construction.

Further research will focus on interaction and modularization concepts. A richer spectrum of interaction concepts is needed, including asynchronous directed interaction. Investigations towards extending the theory to asynchronous communication have been done in (Denker *et al.* 1996). A modularization concept is needed providing generic building blocks with external and internal interfaces related by reification. As for reification, cf. (Denker 1995).

Acknowledgements The authors are grateful to their colleagues for many dis-

cussions. Special thanks are due to Amílcar Sernadas and Carlos Caleiro for many suggestions and hints concerning the distributed temporal logics. Juliana Küster Filipe did a great job in working out proof details. The work of Peter Hartel has been the starting point to introduce concurrency into TROLL. The remarks of the anonymous referees helped to improve the paper.

Work reported here was partially supported by the EU under ESPRIT BRA ASPIRE 22704 and DFG under Eh75/11-1.

REFERENCES

Abadi, M. and Lamport, L. (1995) Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–533, May.

Bonner, A.J. and Kifer, M. (1996) Concurrency and Communication in Transaction Logic, in *Proc. Joint Int. Conf. and Symp. on Logic Programming (JICSLP96), Bonn, Germany* (ed. Maher, M.). The MIT Press.

Caleiro, C. (1996) *Personal communication.*

Clavel, M. and Eker, S. and Lincoln, P. and Meseguer, J. (1996) Principles of Maude, in *Rewriting Logic and its Applications, First International Workshop, Asilomar Conference Center, Pacific Grove, Ca, September 3-6, 1996* (ed. Meseguer, J.), pages 65–89.

Denker, G. (1995) Transactions in Object-Oriented Specifications, in *Recent Trends in Data Types Specification, Proc. 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, S.Margherita, Italy, May/June 1994, Selected papers* (eds. Astesiano, E. and Reggio, G. and Tarlecki,A.), pages 203–218, Springer, Berlin, LNCS 906.

Denker, G. and Küster Filipe, J. (1996) Towards a Model for Asynchronously Communicating Object, in *Proc. 2nd Int. Baltic Workshop on Databases and Information Systems, Tallinn, June 12-14, 1996* (eds. Haav, H.-M. and Thalheim, B.), pages 182–193. Institute of Cybernetics.

Ehrich, H.-D. (1996) *Object Specification*. Technical Report, Technische Universität Braunschweig, `URL://www.cs.tu-bs.de/idb/welcome_e.html`.

Ehrich, H.-D. and Sernadas, A. (1995) Local Specification of Distributed Families of Sequential Objects, in *Recent Trends in Data Types Specification, Proc. 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, S.Margherita, Italy, May/June 1994, Selected papers* (eds Astesiano, E. and Reggio, G. and Tarlecki,A.), pages 219–235. Springer, Berlin, LNCS 906.

Ehrich, H.-D. and Caleiro, C. and Sernadas, A. and Denker, G. (1997) Logics for Specifying Concurrent Information Systems, *forthcoming*.

Hartel, P. (1996) *Konzeptionelle Modellierung von Informationssystemen als verteilte Objektsysteme*. Reihe DISDBIS, infix-Verlag, Sankt Augustin.

Jungclaus, R. and Saake, G. and Hartmann, T. and Sernadas, C. (1995) TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, April.

Krone, M. and Kowsari, M. and Hartel, P. and Denker, G. and Ehrich, H.-D. (1996) Developing an Information System Using TROLL: an Application Field Study, in *Proc. 8th Int. Conf. on Advanced Information Systems Engineering (CAiSE'96)* (eds. Constantopoulos, P. and Mylopoulos, J. and Vassiliou, Y.), pages 136–159, Springer, Berlin, LNCS 1080.

Lodaya, K. and Ramanujam, R. and Thiagarajan, P.S. (1992) Temporal Logics for Communicating Sequential Agents. *Int. Journal of Foundations of Computer Science*, 3(2):117–159.

Meseguer, J. (1992) Conditional Rewriting Logic an a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155.

Meseguer, J. (1993) A Logical Theory of Concurrent Objects and its Realization in the Maude Language, in *Research Directions in Concurrent Object-Oriented Programming* (eds. Agha, G. and Wegner, P. and Yonezawa, A.), pages 314–390. The MIT Press.

Sassone, V. and Nielsen, M. and Winskel, G. (1993) A Classification of Models for Concurrency, in *CONCUR'93, Proc. 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 1993* (ed. Best, E.), pages 325–392. Springer, LNCS 715.

Sernadas, A. and Ramos, J. (1994) The GNOME Language: Syntax, Semantics and Calculus. Technical Report, Instituto Superior Téchnico (IST), Dept. Mathemática, Av. Roviso Pais, 1096 Lisboa Codex, Portugal.

Vaandrager, F.W. (1989) A simple definition for parallel composition of prime event structures. Technical Report CS-R8903, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.

Winskel, G. and Nielsen, M. (1995) Models for Concurrency, in *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling* (eds. Abramsky, S. and Gabbay, D.M. and Maibaum, T.S.E), pages 1–148. Oxford Science Publications.

Biography

Grit Denker received a degree in mathematics in 1991 and received her PhD in 1995 at Technical University of Braunschweig, Germany. Since 1996 she is assistant professor at TU Braunschweig. Her interests are in object-oriented specification of distributed systems, formal methods and models for system specification, concurrency, reification.

Hans-Dieter Ehrich studied mathematics at the University of Kiel, Germany. He received his PhD in 1970 at the University of Hannover, Germany. Since 1982 he is professor at the Technical University of Braunschweig. He leads the database group. His interests are information systems design from theory to applications.