Logics for Databases and Information Systems

edited by
Jan Chomicki and Gunter Saake

Kluwer Academic Publishers, Boston 1998

pages 167–198

Contents

6 LOGICS FOR SPECIFYING CONCURRENT INFORMATION SYSTEMS

Hans-Dieter Ehrich, Carlos Caleiro,
Amilcar Sernadas, and Grit Denker

Abstract: This chapter concentrates on a challenging problem of information system specification and design, namely how to cope on a high level of abstraction with concurrent behaviour and communication as implied by distribution. Since distributed information systems are reactive and open systems maintaining data bases and applications, it is crucial to develop high-level specification techniques that can cope with data and programs as well as with concurrent workflow and communication issues. Techniques from conceptual modeling, abstract data types, concurrent processes and communication protocols are relevant and have to be combined. In the approach presented here, temporal logic is used for specifying sequential object behaviour, and communication facilities are added for specifying interaction between concurrent objects. We study two distributed temporal logics dealing with communication in two different ways. D_0 adds basic statements that can only express synchronous "calling" of predicates, while D_1 adds much richer facilities for making local statements about other objects in their respective local logics. D_0 is more operational and can be animated or implemented more easily, while D_1 is intuitively more appealing and convenient for modeling and specification. We demonstrate by example how D_1 can be effectively reduced to D_0 in a sound and complete way.

This work was partially supported by the PRAXIS XXI Program and JNICT, as well as by PRAXIS XXI Projects 2/2.1/MAT/262/94 SitCalc, PCEX/P/MAT/46/96 ACL plus 2/2.1/TIT/1658/95 LogComp, and ESPRIT IV Working Groups 22704 ASPIRE and 23531 FIREworks.

6.1 INTRODUCTION

In the early phases of information systems development, it is essential to work on a high level of abstraction: careful conceptual modeling and specification techniques help making the right design decisions and adapting the system to changing needs. The objective is to give the developer the ability to prescribe the properties of a system, and to predict and check its behaviour by reasoning, simulation and animation based on specification, and to give a sound reference basis for testing the implementation.

This chapter is about high-level specification techniques for *distributed* information systems, giving due attention to concurrency and communication among sites. While implementation platforms like CORBA are evolving to facilitate implementation, little is known about how to set up and specify distributed data and behaviour models in a meaningful way.

Information systems are reactive systems with the capability of maintaining and utilizing large amounts of data. A crucial point for specification is to choose the right logic—or family of logics. Our approach combines ideas and concepts from the object-oriented systems view, and from the traditions of conceptual modeling, behaviour modeling, abstract data type theory, specification of reactive systems, and concurrency theory. It is based on experiences with developing the OBLOG family of languages and their semantic foundations that started with [SSE87], in particular the TROLL and GNOME object specification languages. References are given in section 6.7, together with an account of related work.

The outline of the paper is as follows. In section 6.2, we introduce basic concepts and ideas by means of an example. In section 6.3, we give an account of the local propositional logic L used for specifying single objects in isolation. In section 6.4, two distributed propositional logics are introduced that add communication facilities to L: D_0 adds basic statements that can only express synchronous "calling" of predicates, while D_1 adds fancy facilities for making local statements about other objects in their respective local logics. D_0 is more operational and can be animated or implemented more easily, while D_1 is intuitively more appealing and convenient for modeling and specification. In section 6.5, we demonstrate by example how D_1 can be effectively reduced to D_0. Thus, D_1 does not have more expressive power than D_0, and D_1 specifications may be automatically translated to D_0 descriptions. In section 6.6, we give an extended example drawn from a real application that shows how convenient it is to use D_1. Hints to work related to our approach are compiled in section 6.7. The chapter closes with concluding remarks in section 6.8.

D_1 is especially useful for modeling object behaviour and workflow involving inter-object communication in distributed information systems. Data modeling is captured as an integral part of object modeling. Our specification techniques are also useful for coping with interoperability among heterogeneous information systems, and for integrating legacy systems into new products. We do not envisage, however, to use our logics for querying.

No method for using D_1 and D_0 is described in this chapter, and no tools either for working with the logics. Work is in progress for animating and analyzing TROLL specifications that are based on D_0. We conceive to use D_1 for specifying properties of distributed information systems, and to validate the design partly by reasoning about the D_1 specification, partly by generating test cases for validating the implementation against the specification, and partly by translation to D_0 and using D_0-based tools for animation, reasoning, model checking, or testing. Reasoning may often be done by hand in a semantic, i.e., traditional mathematical way rather than using proof systems. Proof systems may have their value but this is not our current emphasis. The same holds for attempts to generate implementations from specifications.

6.2 OVERVIEW

In sufficiently abstract view, an information system is a collection of objects operating concurrently and interacting by exchanging messages. An object is an encapsulated unit of structure and behaviour. It has an identity which persists through change.

The operations of an object are usually called *methods*. In the object model of object-oriented programming, a method may change state and deliver a value. This model also underlies object specification languages like FOOPS and ETOILE (cf. section 6.7), and it corresponds with the core model of ODMG [Cat94]. The object model of some object-oriented databases is more restricted; it separates state-changing *proper* methods from side-effect free read methods called *attributes*; the latter appear in the extended model of ODMG. This model also underlies TROLL, we adopt it here.

We illustrate the concept by means of an example, namely state variables in the sense of imperative programming. For specification, we use an ad-hoc notation that is close in spirit to TROLL and GNOME but uses traditional logic notation instead of any of their concrete syntax.

Example 1 Here is the formal specification of a class $\mathtt{Var}[s]$ of state variables retaining values of data sort s. Let i be such a state variable. i has an attribute $i.\mathtt{val}$ of sort s denoting its current value. It has actions $i.\mathtt{alloc}$ for allocating memory, i.e., creating the variable, $i.\mathtt{assign}(s)$ for assigning values (i.e., an action $i.\mathtt{assign(v)}$ for every value $\mathtt{v}$ of sort s), and $i.\mathtt{free}$ for giving the variable back to free space, i.e., deleting it.

In what follows, we specify the *class* of state variables, i.e., an unnamed generic state variable with attribute `val` and actions `alloc`, `assign(`s`)` and `free`. In the axioms, we use the *until* temporal operator $\varphi\,\mathcal{U}^{\circ}\,\psi$ for expressing that φ holds from now on until ψ holds for the next time. If a is an action symbol, we also use a as a predicate symbol expressing that action a has occurred in arriving at the current state. The notation $\triangleright a$ means that action a is enabled, i.e., may occur in a transition from the current state. So the $\triangleright$ predicate highlights the "menu" of actions that may be chosen to proceed; $\triangleright a$ is a *precondition* for a to occur. Note that in a state transition caused by action a, $\triangleright a$ holds before and a holds after the transition.

class $\mathtt{Var}[s]$;
  attribute $\mathtt{val}{:}\,s$;
  actions $\mathtt{alloc};\ \mathtt{assign}(s);\ \mathtt{free}$;
  axioms $\mathtt{v,w}{:}\,s$;
    $\mathtt{alloc} \Rightarrow (\neg\triangleright\mathtt{alloc} \wedge \triangleright\mathtt{assign(v)} \wedge \triangleright\mathtt{free}\ )\ \mathcal{U}^{\circ}\ \mathtt{free},$
    $\mathtt{free} \Rightarrow \neg\triangleright\mathtt{alloc} \wedge \neg\triangleright\mathtt{assign(v)} \wedge \neg\triangleright\mathtt{free},$
    $\mathtt{assign(v)} \Rightarrow \mathtt{val{=}v}\ \mathcal{U}^{\circ}\ (\mathtt{assign(w)} \vee \mathtt{free}\ )$
end

The axioms say that (1) after allocation and before deletion, another allocation is disabled but value assignment and deletion are enabled; (2) after deletion, no action is enabled; (3) the value after assignment is retained until the next assignment, or until deletion.

Let $I = \{i, j, \ldots\}$ be object identities for naming state variables. Each identity i denotes a variable instance, with attribute i`.val` and actions i`.alloc`, i`.assign(`s`)` and i`.free`. Its behaviour is given by the local set of axioms obtained from the ones given above by prefixing by $i.$ the corresponding ones of the instance. For instance, $i.(\mathtt{assign(v)} \Rightarrow \mathtt{val{=}v}\,\mathcal{U}^{\circ}(\mathtt{assign(w)} \vee \mathtt{free}))$ is the third local axiom of variable i.

For demonstrating communication, let i and j be two integer variables. We would like to specify that, whenever i is assigned some value n, then j is simultaneously set to 0. Such a situation arises, e.g., when counting sections within chapters; in a new chapter, section counting starts from the beginning.

object system `CommunicatingVariables`;
  objects i,j: `Var[int]`;
  axioms n: `int`;
    $i.(\mathtt{assign}(n) \Rightarrow j.\mathtt{assign}(0))$
end.

The axiom is a local condition for i, it is an instance of *action calling* that is the basic communication mechanism in TROLL and other approaches. In section 6.4, we formalize this idea in the logic D_0.

Now consider the following communication. We would like to express that, if i's value is changed, then j is set to 0 *some time later*. Using the action-calling logic indicated above, we have to introduce new communication actions, say `send` and `receive`. The system communication axiom then reads

$$i.(\texttt{send} \Rightarrow j.\texttt{receive}).$$

In the class axioms, we have to relate the `send` and `receive` actions to value assignment. We discuss three possibilities to make the above idea precise, depending on when communication should take place. F is the temporal *sometime in the future* operator.

$$\begin{array}{llll} 1 & \texttt{assign}(n) \Rightarrow \texttt{send}\,, & \texttt{receive} \Rightarrow \mathsf{F}\,\texttt{assign}(0) \\ 2 & \texttt{assign}(n) \Rightarrow \mathsf{F}\,\texttt{send}\,, & \texttt{receive} \Rightarrow \texttt{assign}(0) \\ 3 & \texttt{assign}(n) \Rightarrow \mathsf{F}\,\texttt{send}\,, & \texttt{receive} \Rightarrow \mathsf{F}\,\texttt{assign}(0) \end{array}$$

In the first solution, communication takes place immediately when the new value is assigned to i; some time later, 0 is assigned to j. In the second solution, communication takes place some time after the new value is assigned to i; immediately at communication time, 0 is assigned to j. In the third solution, communication takes place some time after the new value is assigned to i; some time later, 0 is assigned to j. Of course, the last version covers the other two. In either case, communication is synchronous.

In the fancier logic D_1 to be described in section 6.4, the situation may be described without introducing extra communication actions `send` and `receive`, and with only one axiom for replacing two of the above.

$$\begin{array}{ll} 1 & i.(\texttt{assign}(n) \Rightarrow j.\,\mathsf{F}\,\texttt{assign}(0)) \\ 2 & i.(\texttt{assign}(n) \Rightarrow \mathsf{F}\; j.\texttt{assign}(0)) \\ 3 & i.(\texttt{assign}(n) \Rightarrow \mathsf{F}\; j.\,\mathsf{F}\,\texttt{assign}(0)) \end{array}$$

D_1 is able to talk about communication in an *implicit* way, without having to recur to explicit communication actions. In section 6.5, we show that D_1 can be effectively reduced to D_0; explicit communication actions are introduced systematically in the reduction process from D_1 to D_0. This is useful for translating D_1 specifications to more operational D_0 specifications for which analysis and animation tools are easier to implement.

Note that all communications mentioned so far are synchronous. *Asynchronous* communication may be modeled by letting a message be an object that synchronizes with the sender when sent, and with the receiver when received. It is also possible to make asynchronous communication a primitive concept and treat synchronous communication as a special case.

6.3 LOCAL LOGIC L

The syntax of an object specification is given by its attribute and action symbols, giving rise to state predicates saying which attributes have which values, which actions have occurred, which actions are enabled, etc. We abstract from details and assume that some set of state predicates is given.

Definition 1 *An object signature P is a denumerable set whose elements are called state predicates. An object specification $Ospec = (P, \Phi)$ consists of an object signature P and a set of formulae Φ called behaviour axioms.*

For the behaviour axioms, we may choose among a wide variety of object logics. Our choice is a propositional temporal logic that goes a little beyond linear time and offers a weak kind of branching expressiveness via the *may* operator $\mathcal{M}$. $\mathcal{M}\,\varphi$ expresses that φ may hold, i.e., φ holds in some state that may have been reached from the previous state, including the current one. So a formula φ may only hold if $\mathcal{M}\,\varphi$ holds, i.e., we always have $\varphi \Rightarrow \mathcal{M}\,\varphi$. The $\mathcal{M}$ operator is useful, among others, for defining action enabling, cf. example 1: an action is enabled iff it may have happened in the next state, formally $\triangleright a \Leftrightarrow \mathsf{X}\ \mathcal{M}\, a$, where X is the temporal *next* operator.

For the sake of conciseness, we concentrate on future-directed temporal operators. The corresponding past-directed ones offer more specification convenience but do not increase specification power [LS95].

Definition 2 *The syntax of* L *is given by*

$$\mathsf{L} ::= P \mid \mathit{false} \mid (\mathsf{L} \Rightarrow \mathsf{L}) \mid (\mathsf{L}\ \mathcal{U}\ \mathsf{L}) \mid (\mathcal{M}\ \mathsf{L})$$

The predicates in P are *flexible*, i.e., we intend to give them time-dependent meanings. The other symbols are *rigid*, i.e., we intend to give them time-independent meanings.

false is the usual logical constant, $\Rightarrow$ denotes logical implication, $\mathcal{U}$ is the *until* temporal operator, and $\mathcal{M}$ is the *may* temporal operator. $\varphi\,\mathcal{U}\,\psi$ means that φ will always be true from the next moment on until ψ becomes true for the next time; φ need not be true any more as soon as ψ holds; ψ must eventually become true. $\mathcal{M}\,\varphi$ means that φ may be true in the sense that it is true in the current state or an alternative state that might have been entered from the previous state.

As usual, we introduce derived connectives, e.g., $\neg\,\varphi$ for $\varphi \Rightarrow \mathit{false}$, *true* for $\neg\,\mathit{false}$, $\varphi \vee \psi$ for $\neg\,\varphi \Rightarrow \psi$, etc. We also make use of derived temporal operators like $\mathsf{X}\,\varphi$ for $\mathit{false}\,\mathcal{U}\,\varphi$ expressing *next*, $\mathsf{F}\,\varphi$ for $\varphi \vee \mathit{true}\,\mathcal{U}\,\varphi$ expressing *sometime*, $\mathsf{G}\,\varphi$ for $\neg(\mathsf{F}(\neg\,\varphi))$ expressing *always*, and $\varphi\,\mathcal{U}^\circ\,\psi$ for

$\varphi \wedge \varphi\,\mathcal{U}\,\psi$ expressing *from now on ... until ...* As suggested by action enabling, we may introduce a general form of *formula enabling* by defining that $\triangleright\varphi$ holds iff $\mathsf{X}\,\mathcal{M}\,\varphi$ holds.

Furthermore, we apply the usual rules for omitting brackets.

For interpreting L, we need a model of sequential computation that smoothly extends to a full model of concurrency. Our choice is based on the simple fact that the execution of an object's sequential program leads to a finite or infinite sequence of events. Events are occurrences of actions that change the object's state. At each state, the execution may proceed in several ways. So the set of all possible executions has a natural branching or tree structure. Allowing for several start states, we arrive at a set of trees: a forest or *grove*. The nodes are events, and the edges represent sequencing: $e_1 \to e_2$ means that event e_1 occurs immediately before event e_2. Another way to put it is that e_1 is a precondition for e_2. The model we envisage may be described as an unfolded state transition system.

Let Ev be a set of elements called events, and let $\to$ be a binary relation on Ev.

Definition 3 *An event grove is an acyclic graph $G = (Ev, \to)$ such that, for all events $e_1, e_2 \in Ev$, if there is an event $e_3 \in Ev$ such that $e_1 \to^* e_3$ and $e_2 \to^* e_3$, then $e_1 \to^* e_2$ or $e_2 \to^* e_1$.*

A trace in G is a backward-closed totally ordered set $T \subseteq Ev$ i.e., $e \in T$ and $e' \to e$ imply $e' \in T$. The set of traces in G is denoted by $\mathcal{T}(G)$.

A life cycle in G is a maximal trace in G in the sense that it is not properly contained in another trace. The set of life cycles in G is denoted by $\mathcal{L}(G)$.

As a graph, an event grove is a set of rooted trees. A trace is a linear path starting from a root. A life cycle is a trace that is infinite or ends at a leaf. Traces are prefixes of life cycles.

For those who are familiar with event structures as a model of concurrency [Win87; WN95], we note that an event grove $G = (Ev, \to)$ determines a prime event structure $E(G) = (Ev, \to^*, \#)$ in a canonical way: causality is given by the reflexive and transitive closure $\to^*$ of $\to$, and all causally unrelated events are in conflict. Thus, the concurrency relation is empty—which is equivalent to saying that $E(G)$ is sequential.

Event groves may also be seen as Kripke structures, the standard interpretation structures for modal logics. If $G = (Ev, \to)$ is an event grove, then Ev is the set of possible worlds, $\to$ is an accessibility relation corresponding to *next*, and the reflexive and transitive closure $\to^*$ is an accessibility relation corresponding to *eventually*.

In order to provide for interpretation structures, each event is labelled by an object state represented by the set of propositions that hold true at that state.

Definition 4 *Let $G = (Ev, \rightarrow)$ be an event grove and P an object signature. A labelling for G is a total function $\lambda : Ev \rightarrow 2^P$.*

Our denotational models for objects and object classes are labelled event groves. In fact, at this level of abstraction, there is no difference between an object instance and an object class: an object is an isomorphic copy of its class. In order to emphasize the abstract view, we speak of *object behaviours* as generalizations of objects and classes.

Definition 5 *An object behaviour is a labelled event grove, i.e., a triple $B = (G, \lambda, P)$ where $G = (Ev, \rightarrow)$ is an event grove, and $\lambda : Ev \rightarrow 2^P$ is a labelling for G.*

Formulae of the local logic L are interpreted at events in object life cycles.

Definition 6 *Let $B = (G, \lambda, P)$ be an object behaviour and $L \in \mathcal{L}(G)$ a life cycle in G. Let $e \in L$ be an event and $p \in P$ a state predicate.*

The satisfaction relation $\models$ is inductively defined as follows.

$B, L, e \models p$	*iff* $p \in \lambda(e)$;
$B, L, e \models \mathit{false}$	*does not hold;*
$B, L, e \models (\varphi \Rightarrow \psi)$	*iff* $B, L, e \models \varphi$ *implies* $B, L, e \models \psi$;
$B, L, e \models (\varphi\, \mathcal{U}\, \psi)$	*iff there is a future event* $e' \in L$, $e \rightarrow^+ e'$, *where* $B, L, e' \models \psi$, *and* $B, L, e'' \models \varphi$ *for every event* $e'' \in L$ *such that* $e \rightarrow^+ e'' \rightarrow^+ e'$;
$B, L, e \models (\mathcal{M}\, \varphi)$	*iff* $B, L, e \models \varphi$ *or there are a previous event* $e' \in L$, $e' \rightarrow e$, *a life cycle* L' *in* G *such that* $e' \in L'$, *and a successor event* $e'' \in L'$, $e' \rightarrow e''$, *where* $B, L', e'' \models \varphi$.

Note that the last rule is not redundant: $\mathcal{M}\, \varphi$ must be true at the beginning of a life cycle if φ is true there, otherwise the intended tautology $\varphi \Rightarrow \mathcal{M}\, \varphi$ would not hold there.

Note that, given an object behaviour B, a formula φ may be true in a life cycle L at an event e but may be not true in another life cycle L' at the same event e.

By the abbreviations introduced above, we may derive satisfaction conditions for other connectives and temporal operators, e.g.,

$B, L, e \models (\mathsf{X}\, \varphi)$	holds iff there is a next event $e' \in L$, $e \rightarrow e'$, where $B, L, e' \models \varphi$ holds.

A formula φ is said to be *valid* in life cycle L in B, in symbols $B, L \vDash \varphi$, iff $B, L, e \vDash \varphi$ holds for all events e in L.

We cannot elaborate on the semantics of object specification. Roughly speaking, given an object specification $\mathit{Ospec} = (P, \Phi)$ and an object behaviour $B = (G, \lambda, P)$, the semantics of Ospec is given by the substructure $[\![\mathit{Ospec}]\!] \subseteq B$ consisting of all life cycles in B in which all axioms in Φ are valid. The question is how to define a canonical object behaviour B from the specification. The interested reader may find some hints in [ES95].

6.4 DISTRIBUTED LOGICS

A system is a collection of interacting objects. In what follows, we assume that we have a fixed finite set of objects, represented by their identities $I = \{1, \ldots, n\}$. In order to emphasize distribution of objects, the identities in I are also called *localities*. Each object $i \in I$ has its own local logic L^i with its own local set of state predicates P_i.

Definition 7 *A system signature is a pair* $\mathsf{P} = (I, P)$ *such that* $P = \{P_1, \ldots, P_n\}$ *is an I-indexed family of sets of local state predicates. A system specification* $\mathit{Sspec} = (\mathsf{P}, \Phi)$ *consists of a system signature* P *and a set of formulae* Φ *in a distributed logic.*

In this section, we introduce two distributed logics, D_0 offering only poor communication facilities but being operational, and D_1 offering fancy facilities for expressing communication in an implicit way. Both logics are propositional.

For interpreting both logics, *distributed* event groves provide suitable structures. Informally speaking, a distributed event grove is a family of event groves that may share events, i.e., the local event sets need not be disjoint. This is our denotational system model.

Definition 8 *Let I and Ev be given sets of identities and (global) events, respectively. A distributed event grove over I is an I-indexed family $G = \{G_1, \ldots, G_n\}$ where $G_i = (Ev_i, \rightarrow_i)$ is an event grove, and $Ev_i \subseteq Ev$ for every $i \in I$.*

For readers familiar with event structures [Win87; WN95], we note that a distributed event grove may be considered as a presentation of a prime event structure $E(G) = (Ev, \rightarrow^*, \#)$ where $Ev = \bigcup_{i \in I} Ev_i$, $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow = \bigcup_{i \in I} \rightarrow_i$, and conflict is given by $\# = \bigcup_{i \in I} \#_i$ where $\#_i$ is the conflict relation at locality i. Thus, any two events at different localities that are not in causal relationship are concurrent. Note that $E(G)$ is in general not sequential but truly concurrent. In fact, any labelled prime event structure can be presented as a distributed event grove.

Definition 9 *Let $G = \{G_1, \ldots, G_n\}$ be a distributed event grove over I.*

A distributed trace in G is a set $T \subseteq Ev$ such that $T_i = \{e \in T \mid e \in Ev_i\}$ is a trace in G_i for every $i \in I$.

A distributed life cycle in G is a set $L \subseteq Ev$ such that $L_i = \{e \in L \mid e \in Ev_i\}$ is a life cycle in G_i for every $i \in I$.

Intuitively, a distributed trace is a web of local traces glued together at interaction events. The same holds for distributed life cycles. A distributed life cycle is a maximal distributed trace in the sense that it is not properly contained in another distributed trace.

Our denotational model for an object system is a *labelled* distributed event grove called system behaviour.

Definition 10 *A system behaviour over P is a triple $B = (G, \lambda, \mathsf{P})$ where $G = \{G_1, \ldots, G_n\}$ is a distributed event grove over I, and $\lambda = \{\lambda_1, \ldots, \lambda_n\}$ is an I-indexed family of labellings such that $B_i = (G_i, \lambda_i, P_i)$ is an object behaviour for every $i \in I$.*

Note that shared events have several labels, one for each object sharing the event.

Interpretation structures for both D_0 and D_1 are pairs (B, L) where $B = (G, \lambda, P)$ is a system behaviour and $L \in \mathcal{L}(G)$ is a distributed life cycle in G.

Distributed logic D_0

Let $\mathsf{P} = (I, P)$ be a system signature, $P = \{P_1, \ldots, P_n\}$. In each of the local sets of state predicates P_i, we distinguish a subset $C_i \subseteq P_i$ of *communication predicates*. The intuitive meaning is that communication predicates are "visible by other objects". For instance, in the Troll and Gnome languages, an action occurrence may have a global effect by calling an action in another object, whereas action enablings and values of attributes are not seen by other objects.

Definition 11 *The syntax of D_0 is given by*

$$\mathsf{D}_0 ::= \mathsf{D}_0^1 \mid \ldots \mid \mathsf{D}_0^n$$

For each object $i \in I$, we have

$$\mathsf{D}_0^i ::= i.\mathsf{L}_0^i \mid i.\mathsf{CC}_0^i$$
$$\mathsf{L}_0^i ::= @I \mid P_i \mid \mathit{false} \mid (\mathsf{L}_0^i \Rightarrow \mathsf{L}_0^i) \mid (\mathsf{L}_0^i \,\mathcal{U}\, \mathsf{L}_0^i) \mid (\,\mathcal{M}\, \mathsf{L}_0^i)$$
$$\mathsf{CC}_0^i ::= (C_i \Rightarrow 1.C_1) \mid \ldots \mid (C_i \Rightarrow n.C_n)$$

D_0^i is the logic at locality i with callings to other localities. This is the operational distributed logic underlying Troll3: local axioms are labelled by localities, and the only formulae across localities are action callings.

The predicates $@I$ in the local logics need explanation. The intuitive meaning of $i.@j$ is that i "talks to" j, i.e., i synchronizes with j and shares an event with j. This expresses that there is *some* communication, as opposed to the specific communications described by the communication formulae: $i.(c \Rightarrow j.c')$ says that whenever c is true for i, then i synchronizes with j at an event where c' is true for j. Here are the formal details.

Definition 12 *Let I be a set of object identities, $B = (G, \lambda, P)$ a system behaviour with local behaviours $B_i = (G_i, \lambda_i, P_i)$, $i \in I$, and $L \in \mathcal{L}(G)$ a distributed life cycle in G with local life cycles $L_i \subseteq L$, $i \in I$. Let $e \in L$ be an event, $p \in P$ a state predicate, $i, j \in I$ identities, and $c \in C_i$, $c' \in C_j$ communication predicates.*

The satisfaction relation $\models_0$ for D_0 is defined by

$$B, L \models_0 i.\varphi \qquad \textit{iff} \quad B, L, e \models_0^i \varphi \textit{ holds for every } e \in L_i.$$

For each $i \in I$, the relation $\models_0^i$ is inductively defined as follows

$$\begin{array}{ll}
B, L, e \models_0^i @j & \textit{iff } e \in L_j; \\
B, L, e \models_0^i p & \textit{iff } p \in \lambda_i(e); \\
B, L, e \models_0^i \textit{false} & \textit{does not hold;} \\
B, L, e \models_0^i (\varphi \Rightarrow \psi) & \textit{iff } B, L, e \models_0^i \varphi \textit{ implies } B, L, e \models_0^i \psi; \\
B, L, e \models_0^i (\varphi\, \mathcal{U}\, \psi) & \textit{iff there is a future event } e' \in L_i,\ e \rightarrow_i^+ e', \\
& \textit{where } B, L, e' \models_0^i \psi, \textit{ and } B, L, e'' \models_0^i \varphi \\
& \textit{for every event } e'' \in L_i \textit{ such that } e \rightarrow_i^+ e'' \rightarrow_i^+ e'; \\
B, L, e \models_0^i (\mathcal{M}\, \varphi) & \textit{iff } B, L, e \models_0^i \varphi \textit{ or there are a previous event } e' \in L_i, \\
& e' \rightarrow_i e, \textit{ a distributed life cycle } L' \textit{ in } G \textit{ such that} \\
& e' \in L', \textit{ and a successor event } e'' \in L'_i,\ e' \rightarrow_i e'', \\
& \textit{where } B, L', e'' \models_0^i \varphi; \\
B, L, e \models_0^i (c \Rightarrow j.c') & \textit{iff } B, L, e \models_0^i c \textit{ implies } e \in Ev_j \textit{ and } B, L, e \models_0^j c'.
\end{array}$$

Except for the first and last rules, local satisfaction $\models_0^i$ of D_0 formulae in distributed life cycles is defined the same way as satisfaction $\models$ of L in local life cycles, cf. definition 6: just replace $\models$ there by $\models_0^i$.

The first rule defines the locality predicate $@j$ that has been intuitively explained above: communication is modelled by shared events. The last rule formalizes the basic communication mechanism of "predicate calling": $i.(c \Rightarrow j.c')$ holds iff validity of c in i at event e implies validity of c' in j at the same shared event e.

Example 1 in section 6.2 illustrates the use of D_0:

$$i.(\texttt{assign}(n) \Rightarrow j.\texttt{assign}(0))$$

means that if value n is assigned to variable i, then value 0 is assigned to variable j at the same time. More examples of D_0 formulae are given in the next section.

Distributed logic D_1

For D_1, no special communication predicates are introduced because *all* local formulae of an object are "visible by other objects". The idea is that local statements about another object in the local language of the latter can be made in any object.

Definition 13 *The syntax of D_1 is given by*

$$\mathsf{D}_1 ::= \mathsf{D}_1^1 \mid \ldots \mid \mathsf{D}_1^n$$

For each object $i \in I$, we have

$$\begin{aligned}
&\mathsf{D}_1^i ::= i.\mathsf{L}_1^i\\
&\mathsf{L}_1^i ::= P_i \mid \mathit{false} \mid (\mathsf{L}_0^i \Rightarrow \mathsf{L}_0^i) \mid (\mathsf{L}_0^i \,\mathcal{U}\, \mathsf{L}_0^i) \mid (\,\mathcal{M}\, \mathsf{L}_0^i) \mid \mathsf{CC}_1\\
&\mathsf{CC}_1 ::= \mathsf{D}_1
\end{aligned}$$

D_1^i is the logic at locality i which allows for local statements about any other locality j in its local logic D_1^j. Note that D_1 does not have an explicit locality predicate, it is definable by $@i$ iff $i.\mathit{true}$. Taking this into account, every D_0 formula is a D_1 formula, i.e., D_0 is a sublogic of D_1, $\mathsf{D}_0 \subseteq \mathsf{D}_1$.

Definition 14 *Let I be a set of object identities, $B = (G, \lambda, P)$ a system behaviour with local behaviours $B_i = (G_i, \lambda_i, P_i)$, and $L \in \mathcal{L}(G)$ a distributed life cycle in G with local life cycles $L_i \subseteq L$. Let $e \in L$ be an event, $p \in P$ a state predicate, and $i, j \in I$ identities.*

The satisfaction relation $\vDash_1$ for D_1 is defined by

$B, L \vDash_1 i.\varphi$	*iff* $B, L, e \vDash_1^i \varphi$ *holds for every* $e \in L_i$.

For each $i \in I$, the relation $\vDash_1^i$ is inductively defined as follows

$B, L, e \vDash_1^i p$	*iff* $p \in \lambda_i(e)$;
$B, L, e \vDash_1^i \mathit{false}$	*does not hold;*
$B, L, e \vDash_1^i (\varphi \Rightarrow \psi)$	*iff* $B, L, e \vDash_1^i \varphi$ *implies* $B, L, e \vDash_1^i \psi$;
$B, L, e \vDash_1^i (\varphi \,\mathcal{U}\, \psi)$	*iff there is a future event* $e' \in L_i$, $e \rightarrow_i^+ e'$, *where* $B, L, e' \vDash_1^i \psi$, *and* $B, L, e'' \vDash_1^i \varphi$ *for every event* $e'' \in L_i$ *such that* $e \rightarrow_i^+ e'' \rightarrow_i^+ e'$;
$B, L, e \vDash_1^i (\,\mathcal{M}\, \varphi)$	*iff* $B, L, e \vDash_1^i \varphi$ *or there are a previous event* $e' \in L_i$, $e' \rightarrow_i e$, *a distributed life cycle* L' *in* G *such that* $e' \in L'$, *and a successor event* $e'' \in L'_i$, $e' \rightarrow_i e''$, *where* $B, L', e'' \vDash_1^i \varphi$;

$$B, L, e \models_1^i j.\varphi \qquad \textit{iff } e \in Ev_j \textit{ and } B, L, e \models_1^j \varphi.$$

Local satisfaction $\models_1^i$ of D_1 formulae in distributed life cycles is defined in the same way as satisfaction $\models$ of L in local life cycles, cf. definition 6: just replace $\models$ there by $\models_1^i$ to obtain the first five of the above rules. So these rules also correspond closely with D_0 satisfaction, i.e., rules two to six in definition 12.

The last rule defines all the convenience that D_1 offers, namely to include arbitrary statements about other objects. These statements may refer to other objects in turn, i.e., references to other objects may be nested in an arbitrary way.

The following examples demonstrate some of the convenience to express communication patterns; we give D_1 formulae along with informal natural language translations where the time unit for the temporal *next* operator is assumed to be one day.

Example 2 Let $i, u, j \in I$ *("I, you, Jim").*

$i.(\,@u \wedge u.\mathsf{X}\,\varphi)$	I talk to you and you tell me that you expect φ tomorrow.
$i.u.\mathsf{X}\,\varphi$	you tell me that you expect φ tomorrow (equivalent to previous one).
$i.(@u \wedge u.\mathsf{X}\,@\,j)$	I talk to you and you tell me that you will contact Jim tomorrow.
$i.u.\mathsf{X}\,@\,j$	you tell me that you will contact Jim tomorrow (equivalent to previous one).
$i.\,\mathsf{G}(\,@u \;\Rightarrow\; \mathsf{X}\,\varphi)$	whenever I talk to you, I have φ the next day.
$i.(\varphi \Rightarrow \mathsf{X}\,@\,u)$	if φ holds, then I talk to you the next day.
$i.(\varphi \Rightarrow u.\mathsf{X}\,\psi)$	if φ holds, then you tell me that ψ will hold for you tomorrow.
$i.\,\mathsf{X}\,u.\mathsf{F}\,\varphi$	tomorrow you will tell me that you will sometime enjoy φ.

That the first two formulae have the same meaning can be derived as follows.

$$i.(\,@u \wedge u.\,\mathsf{X}\,\varphi) \;\Leftrightarrow\; i.(u.\mathit{true} \wedge u.\,\mathsf{X}\,\varphi) \;\Leftrightarrow\; i.u.(\mathit{true} \wedge \mathsf{X}\,\varphi) \;\Leftrightarrow\; i.u.\,\mathsf{X}\,\varphi$$

A similar argument demonstrates equivalence of the third and fourth formulae.

6.5 REDUCTION

We show by a more elaborate and illustrative example that D_1 can deal with intricate high level interaction requirements in a rather simple way. We then illustrate how any D_1 specification can be translated into an equivalent D_0 specification. The details as well as a proof of soundness and completeness can be found in [CE98].

Example 3 Consider a system consisting of two objects, SENDER (s) and RECEIVER (r) such that

(1) SENDER has an attribute *val* which determines the value v it may send;
(2) RECEIVER has an attribute *var* which is updated with the value v received;
(3) if SENDER sends a value v, then v will eventually be communicated to RECEIVER who will then eventually receive v;
(4) if RECEIVER receives v, then it will eventually communicate to SENDER an acknowledgment of receipt.

Concentrating on interactions between SENDER and RECEIVER, the situation is illustrated in figure 6.1.

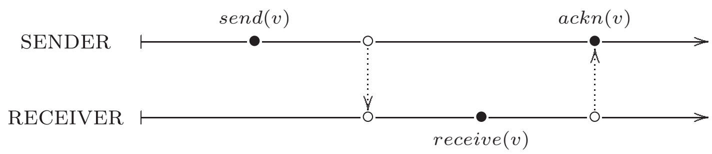

Figure 6.1 Communication between SENDER and RECEIVER.

The system signature is $\mathsf{P} = (\{s, r\}, \{P_s, P_r\})$ where the SENDER and RECEIVER parts are given as follows. V is a given set of values.

SENDER $\quad P_s ::= send(V) \mid ackn(V) \mid val = V$
RECEIVER $\quad P_r ::= receive(V) \mid var := V$

Here is a D_1 specification of the system.

SENDER *behaviour*

s11 $\quad s.(val = v \wedge val = w) \Rightarrow v = w$, for any $v, w \in V$;
s12 $\quad s.(\triangleright\, send(v) \Rightarrow val = v)$, for each $v \in V$;
s13 $\quad s.(send(v) \Rightarrow (\mathsf{F}\, r.(\mathsf{F}\, receive(v))))$, for each $v \in V$;

Axiom s11 says that the SENDER's value is unique; axiom s12 is an enabling condition: only the current value may be sent; axiom s13 specifies that the actual communication takes place sometime between the SENDER's *send* action and the RECEIVER's *receive* action.

RECEIVER *behaviour*

r11 $\quad r.(var := v \wedge var := w) \Rightarrow v = w$, for any $v, w \in V$;
r12 $\quad r.(receive(v) \Rightarrow var := v)$, for each $v \in V$;

r13 $r.(receive(v) \Rightarrow (\mathsf{F}\, s.ackn(v)))$, for each $v \in V$.

Axiom r11 says that only one value may be assigned at a time; axiom r12 says that value v is assigned to var as soon as it is received; axiom r13 says that on receiving v, the RECEIVER expects a communication with the SENDER acknowledging receipt of v. The situation is illustrated in figure 6.1.

It is easy to see that the axioms entail $s.(send(v) \Rightarrow (\mathsf{F}\, ackn(v)))$, i.e., when sending a value, SENDER will receive an acknowledgment some time later.

Note that this style of interaction specification does not mention the communication actions explicitly.

Now we show that such communication patterns are specifiable in D_0 action calling style. In fact, introducing new action symbols $c1(v)$ and $c2(v)$, corresponding with the two communication points identified in figure 6.1, we arrive at the following specification. The idea is to introduce new communication predicates to both objects and use them to specify the communication pattern explicitly, cf. figure 6.2. From

$$s.(send(v) \Rightarrow (\mathsf{F}\, r.(\mathsf{F}\, receive(v)))),$$

by introducing the new communication predicate $c_1(v)$ for $r.(\mathsf{F}\, receive(v))$, we obtain

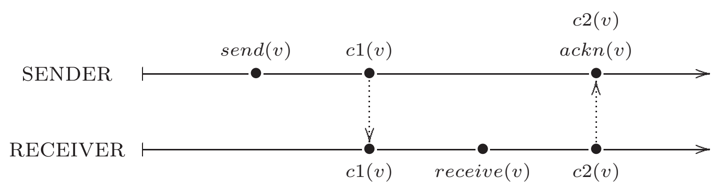

Figure 6.2 Communication between enriched SENDER and RECEIVER.

$s.(send(v) \Rightarrow (\mathsf{F}\, c_1(v))),$
$s.(c_1(v) \Rightarrow r.c_1(v)),$
$r.(c_1(v) \Rightarrow s.c_1(v)),$
$r.(c_1(v) \Leftrightarrow (@s \wedge (\mathsf{F}\, receive(v)))).$

The second and third formulae ensure that $c_1(v)$ expresses a communication. More precisely, for each value v, there are two communication actions $s.c_1(v)$ and $r.c_1(v)$ but only one kind of communication event when both happen together. This expresses synchronous “handshaking” communication. The last formula makes precise that $c_1(v)$ stands for subformula $r.(\mathsf{F}\, receive(v))$ in the context of an s-formula. Note that the $@s$ term is necessary here: without it, the formula would describe unintended behaviour where permanent $c_1(v)$ communication has to hold all the time until $receive(v)$ eventually happens. Analogously, from

$$r.(receive(v) \Rightarrow (\mathsf{F}\ s.ackn(v))),$$

by introducing the new communication predicate $c_2(v)$ for $s.ackn(v)$, we obtain

$$\begin{array}{l} r.(receive(v) \Rightarrow (\mathsf{F}\ c_2(v))), \\ r.(c_2(v) \Rightarrow s.c_2(v)), \\ s.(c_2(v) \Rightarrow r.c_2(v)), \\ s.(c_2(v) \Leftrightarrow (@r \wedge ackn(v))). \end{array}$$

The system signature is $\mathsf{P} = (\{s, r\}, \{P_s \cup C_s, P_r \cup C_r\})$ where the communication predicates are given as follows.

SENDER $C_s ::= c_1(V) \mid c_2(V);$
RECEIVER $C_r ::= c_1(V) \mid c_2(V).$

Summing up what has been discussed above, and integrating the unchanged parts of the D_1 specification, the D_0 specification of the system is as follows.

SENDER *local behaviour*

s01 $s.(val = v \wedge val = w) \Rightarrow v = w$, for any $v, w \in V$;
s02 $s.(\triangleright send(v) \Rightarrow val = v)$, for each $v \in V$;
s03 $s.(send(v) \Rightarrow (\mathsf{F}\ c_1(v)))$, for each $v \in V$;
s04 $s.(c_2(v) \Leftrightarrow (@r \wedge ackn(v)))$, for each $v \in V$;

SENDER *calls*

s05 $s.(c_1(v) \Rightarrow r.c_1(v))$, for each $v \in V$;
s06 $s.(c_2(v) \Rightarrow r.c_2(v))$, for each $v \in V$;

RECEIVER *behaviour*

r01 $r.(var := v \wedge var := w) \Rightarrow v = w$, for any $v, w \in V$;
r02 $r.(receive(v) \Rightarrow var := v)$, for each $v \in V$;
r03 $r.(receive(v) \Rightarrow (\mathsf{F}\ c_2(v)))$, for each $v \in V$;
r04 $r.(c_1(v) \Leftrightarrow (@s \wedge (\mathsf{F}\ receive(v))))$, for each $v \in V$;

RECEIVER *calls*

r05 $r.(c_1(v) \Rightarrow s.c_1(v))$, for each $v \in V$;
r06 $r.(c_2(v) \Rightarrow s.c_2(v))$, for each $v \in V$.

The idea is quite general: for both SENDER and RECEIVER, the first three D_0 *behaviour* axioms result from uniformly replacing subformulae of another locality in the D_1 axioms by explicit communication symbols. Axioms s04 and r04 give definitions for these new symbols.

Note that communication symbols are introduced pairwise, one for each of the two communicating objects. Each pair of new communication symbols is synchronized by defining mutual calling in the SENDER and RECEIVER *calls* axiom pairs s05-r05 and s06-r06.

The reader is invited to convince himself that axioms s01 to s06 and r01 to r06 indeed entail $s.(send(v) \Rightarrow (\mathsf{F}\, ackn(v)))$.

By repeatedly applying steps as suggested by the example, we obtain a D_0 specification from any D_1 specification. Therefore, although D_1 looks more powerful than D_0 at first glance, this is not really true. Indeed, D_1 and D_0 have the same expressive power.

Let $\vartheta : \mathsf{D}_1 \to \mathsf{D}_0$ be the translation outlined above. We extend ϑ to system signatures: $\vartheta(\mathsf{P})$ denotes the system signature obtained from P by adding the extra communication predicates as introduced in the translation process. Let (P, Φ) be a D_1 system specification and $\varphi \in \mathsf{D}_1$. Let $\vDash_1$ denote logical entailment: $(\mathsf{P}, \Phi) \vDash_1 \varphi$ means that φ holds in every system behaviour over signature P that satisfies all formulae in Φ.

The main result is that $\vartheta : \mathsf{D}_1 \to \mathsf{D}_0$ is a sound and complete reduction. More precisely, we have the following.

Theorem 1 *With the items as defined above, we have*

$$(\mathsf{P}, \Phi) \vDash_1 \varphi \quad \textit{iff} \quad (\vartheta(\mathsf{P}), \vartheta(\Phi)) \vDash_1 \varphi \ .$$

The if part formalizes soundness, and the only-if part formalizes completeness. The reader is referred to [CE98] for a detailed account of the reduction and a full worked proof. Note that the formula φ is not translated and entailment is in D_1 throughout. Here we use the fact that $\mathsf{D}_0 \subseteq \mathsf{D}_1$.

Theorem 1 is a corollary of the fact that the reduction from D_0 to D_1 fulfils the condition of a simple map of logics [Mes92] whose semantic translation is a natural isomorphism.

6.6 EXTENDED EXAMPLE

The example is about business process modeling and is extracted from a national German project in which the first and last authors are involved. A brief description of this real-life project is necessary in order to get a feeling how useful D_1 is.

In this project, an information system is being designed and implemented using formal object-oriented techniques from the very beginning (see [KKH+96; HDK+97]). The project is located in the area of computer-aided testing and certifying (CATC) of physical devices. The objective of the information system is to support the various activities of user groups in the PTB (Physikalisch Technische Bundesanstalt) in Braunschweig, the German federal institute of weights and measures. This application is distributed in a natural way because it involves several persons acting concurrently and communicating with each other. About 100 employees will use the system. Thus, the complexity of the organization and the system that is supposed to support this organization is rather high. In order to understand the complex communication structure between the persons involved in the overall business process, we have to build an abstract model of the data flow and the workflow. In what follows, we briefly introduce the problem domain and then turn to a formal specification of the workflow and communication patterns between the persons. We use D_1 as our specification logic.

The project is a cooperation with group 3.5 within PTB, named 'explosion protected electrical equipment'. This group is concerned with testing and certifying explosion proof electrical equipment such as motors, switches, etc., so that it is allowed to be operated in hazardous areas. The assessment procedure consists of testing the formal and informal documents, checking the design papers (mostly technical drawings), and the tests which are carried out. The group is divided into three subgroups dealing with experimental tests in the laboratories, basic administration work, and design approval, respectively (cf. figure 6.3).

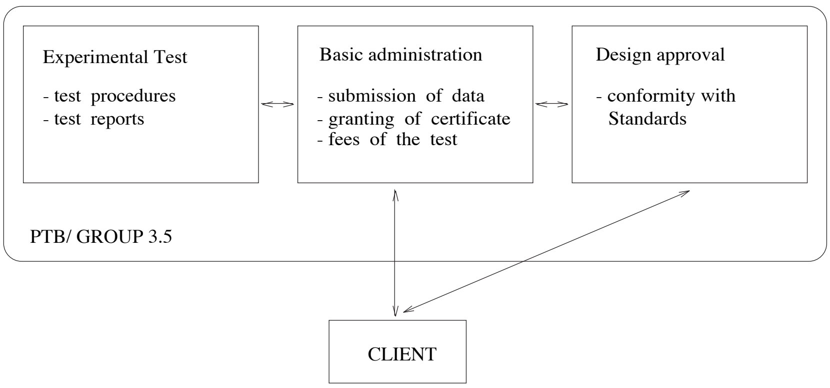

Figure 6.3 Communication structure inside of group 3.5 and with client

Clients contact group 3.5 and ask for the certification of a specific electrical device. Such an application triggers a business process in group 3.5 which involves people from all subgroups. The overall procedure is established due to certain communication constraints. Administration employees talk to people from both other subgroups, employees in the laboratory and officers in charge of the design approval. They also communicate with the client when a certification query is issued to group 3.5. In some cases, employees from the design approval subgroup directly contact the client to ask for missing formal documents, technical drawings, etc. Figure 6.3 gives an overview of the communication relations.

In particular, the following actions take place during the business process of certifying an electrical device. A client may request the certification of an electrical device (*cert-req*). This implies that the administration officer orders sometime later appropriate tests at the laboratories (*test-ord*) and design approvals (*appr-ord*). The corresponding subgroups answer these queries by returning the results of the tests (*test-res(b)*) and design approvals (*appr-res(b)*), respectively. Depending on the results, the administration officer decides whether a certificate is issued (*cert-dec*). Often it is the case that some formal documents are missing. In such a case the design approval officer clarifies the situation by asking the client for the missing papers (*clar-req*). The approval officer informs the administration in doing so. No tests will be carried out as long as the design approval papers are not complete. The approval officer informs the administration officer about the completeness of the application papers (*comp-res(b)*).

The business process is formalized using D_1 in the following way. The system consists of four objects, a CLIENT (c), an ADMINISTRATION officer (a), a LABORATORY officer (l), and a DESIGN APPROVAL officer (d). Thus, the obvious system signature is

$$P ::= (\{c, a, l, d\}, \{P_c, P_a, P_l, P_d\}).$$

The sets of local state predicates are given as follows. Let b be a boolean variable, i.e., $b \in \{true, false\}$.

CLIENT	$P_c ::= \textit{cert-req}$
ADMINISTRATION	$P_a ::= \textit{test-ord} \mid \textit{appr-ord} \mid \textit{cert-dec(b)} \mid \textit{dec=b} \mid$ $\textit{answers=0} \mid \textit{answers=1} \mid \textit{answers=2}$
LABORATORY	$P_l ::= \textit{test-res(b)}$
DESIGN APPROVAL	$P_d ::= \textit{clar-req} \mid \textit{comp-res(b)} \mid \textit{appr-res(b)}$

The administration officer has an attribute *dec* which determines the certification decision made by him. Moreover, he has an attribute *answers* to store how many results he already received. I.e., this attribute determines whether both laboratory and design approval officers delivered their results to him (*answers*$=2$), only one of them (*answers*$=1$), or none (*answers*$=0$).

The system specification is $Sspec = (P, \Phi)$ where $\Phi = \{\Phi_i\}_{i \in I}$ is given as follows.

CLIENT *behaviour*

c1 $c.(\textit{cert-req} \Rightarrow (\mathsf{F}\ a.\textit{cert-dec}(b)))$, for some $b \in B$.

If the client asks for certifying an item, then the administration officer will eventually communicate the decision of the procedure to the client.

ADMINISTRATION *behaviour*

a1 $a.(\textit{test-ord} \Rightarrow (\mathsf{F}\ l.\textit{test-res}(b)))$, for any $b \in B$;
a2 $a.((\neg\ \textit{test-ord})\ \mathcal{U}\ d.\textit{comp-res}(\textit{true}))$;
a3 $a.((\neg\ \textit{cert-dec}(b))\ \mathcal{U}(d.\textit{appr-res}(b^{'})))$, for any $b, b^{'} \in B$;
a4 $a.((\neg\ \textit{cert-dec}(b))\ \mathcal{U}(l.\textit{test-res}(b^{'})))$, for any $b, b^{'} \in B$;
a5 $a.((c.\textit{cert-req}) \Rightarrow (\textit{dec} = \textit{true}\ \wedge\ \textit{answers} = 0))$;
a6 $a.((l.\textit{test-res}(b)) \Rightarrow ((\textit{dec} = \textit{dec} \wedge b) \wedge (\textit{answers} = \textit{answers} + 1)))$, for any $b \in B$;
a7 $a.((d.\textit{appr-res}(b)) \Rightarrow ((\textit{dec} = \textit{dec} \wedge b) \wedge (\textit{answers} = \textit{answers} + 1)))$, for any $b \in B$;
a8 $a.((\textit{answers} = 2) \Rightarrow (\mathsf{F}\ \textit{cert-dec}(\textit{dec})))$;
a9 $a.((\neg\ \textit{cert-dec}(b))\ \mathcal{U}(\textit{answers} = 2))$, for any $b \in B$;
a10 $a.((\textit{answers} = n \wedge \textit{dec} = b^{''}) \Rightarrow$ $((\textit{answers} = n \wedge \textit{dec} = b^{''})\ \mathcal{U}\ (l.\textit{test-res}(b^{'}) \vee d.\textit{appr-res}(b))))$, for any $b, b^{'}, b^{''} \in B$;

The first two axioms assure that the administration officer will get a result from the laboratory after he ordered tests, but those tests cannot be carried out unless the design approval officer reported completeness of the application papers. A decision about the application cannot be made until both results, from design approval and laboratory, have been given ($a3$ and $a4$). The fifth axiom assures that the attributes have the right initialization values. $a6$ and $a7$ formalize which effects the results from laboratory or design approval, respectively, have on attributes *answers* and *dec*. They are abbreviations of syntactically correct D_1 formulae. E.g., formula $a6$ is an abbreviation of the following D_1 formula :

$$a.((\textit{dec} = b^{'} \wedge \textit{answers} = n) \Rightarrow (\mathsf{X}\ (l.\textit{test-res}(b)\ \Rightarrow\ (\textit{dec} = b^{'} \wedge b) \wedge \textit{answers} = n + 1))).$$

Axioms $a8$ and $a9$ state that the administration officer will eventually make the decision after he received all results, but not earlier. The last axiom expresses that only the results from the laboratory or the design approval can change the decision. Thus, only two actions can change the value of that attribute. We assume overall frame axioms like, e.g., attributes can only be altered if an action takes place.

DESIGN APPROVAL *behaviour*

d1 $d.(\mathit{appr\text{-}res}(b) \Rightarrow \mathsf{G}\,\neg \mathit{clar\text{-}req})$, for any $b \in B$;
d2 $d.(\mathit{clar\text{-}req} \Rightarrow @a)$.

The design approval officer cannot ask for a clarification request after he decided the approval procedure. If a clarification request is necessary, he has to "inform" the administration officer. By "inform" we mean that there will be a communication between the approval officer and the administration officer as stated in axiom $d2$.

Given this specification, it can be shown, e.g., that a client will not be contacted by a design approval officer for a clarification after he has received the certification decision from the administration officer,

$$c.(\neg(a.\mathit{cert\text{-}dec}(b) \wedge \mathsf{F}\, d.\mathit{clar\text{-}req})).$$

6.7 RELATED WORK

Information systems modeling and design above the abstraction level of relational databases was largely influenced by the Entity-Relationship approach originated by Chen [Che76], and the study of aggregation and generalization structures by Smith and Smith [SS77]. These ideas found their way into many semantic data models, among them quite a few extensions of the ER model. An excellent recent overview of systems development methods, albeit with an emphasis on requirements engineering, is [Wie96].

Recently there has been considerable activity in the area of object-oriented analysis, modeling and design. The Booch [Boo94], OMT [RBP$^+$91] and OOSE methods [Jac92] merged into the universal modeling language UML [FS97] that has been submitted to the Object Management Group to be considered as a standard. Another successful OO analysis method is Fusion [CAB$^+$94].

These methods are informal or semiformal at best. However, they come along with methodological guidelines and graphical notations. They help to make formal languages fit for use, so they do have their benefits in early modeling and design stages. But they are too unprecise and ambiguous when it comes to animation, verification and forecasting of system properties, and when it comes to generating test cases or even implementations from specifications. And they are limited in scope: concurrency and communication issues are not explicitly treated in these methods.

Among the logic-based formal methods, the work reported here is based on experiences with developing the Oblog family of languages and their semantic foundations. Oblog is being developed into a commercial product [Esp93]. In the academic realm, there are several related developments: Troll [HSJ+94; JSHS96; SJH93; SHJE94; HJ95; EH96; Har97], Gnome [SR94], Lcm [FW93] and Albert [DDPW94].

There are other approaches to formal object specification with a sound theoretical basis. The ones most closely related to ours are Foops [GM87; RS92; GS95] and Maude [Mes93]. Foops is based on Obj3 [GW88] which is in turn based on equational logic. Maude is based on rewriting logic that is a uniform model of concurrency [Mes92]. Other language projects working on related ideas are Ooze [AG91] and Etoile [AB95].

In the Oblog family, Troll3 [EH96; Har97] is the first to address problems of concurrency and communication, and to integrate benefits from the informal methods mentioned above. So there is a graphical notation for Troll3 called omTroll that adopts elements from OMT [JWH+94]. Theoretical foundations of concurrency as applied to this approach have been explored in [ES95; Ehr97]. [EH96] gives a brief overview of Troll3 and omTroll and their logic foundations.

In order to model the sequential behavior of objects and the concurrent behavior of object systems, many models of concurrency may be adopted, see [WN95] for an excellent overview. Our model for denotational semantics is based on labelled event structures because they provide an abstract, powerful and elegant approach. The relationship between event structures and other models of concurrency like labelled transition systems and Petri nets is well investigated [WN95].

Two major approaches have been advocated for specifying and reasoning about concurrent and distributed systems. On one hand, there was a systematic study of the relations and equivalences between behaviours of systems that started in [Mil80] and was subsequently pursued by many people working in process algebra [BK84; Hoa85; HM85; BW90; vG90]. On the other hand, the use of modal logics [Gol92] for characterizing properties of systems has evolved from the early works of Floyd [Flo67] and Hoare [Hoa69] on reasoning about programs. Among them we stress dynamic logics [FL79; Har79; Pel87], temporal logics [Pnu77] and, more recently, logics of knowledge [HM90; HZ92].

The distributed logics defined in this paper are based on temporal logic and an extension towards concurrency called n-agent logic. Temporal logic has been succesfully applied to a number of reactive systems specification problems [MP92], it is the simplest logic that can not only deal with safety properties but also with liveness properties [Lam77]. n-agent logic was introduced and developed in [LT87; LMRT91; LRT92; Thi94; Ram96]. An agent corresponds to an object that may be thought of as a site in a distributed system.

There are several distinct approaches to reasoning about concurrency within the framework of temporal logic. The first and simplest accepts the naive view of a concurrent system modelled by nondeterministic interleaving together with adequate assumptions of fairness [GPSS80; Fra86] on the execution of its components. The use of branching temporal logics like *UB* [BAMP81], *CTL* [CE81] or CTL^* [EH83] and even of linear temporal logic for such purposes has been deeply studied [Pnu77; MP92; Wol95; Pen95]. In particular, most of the work on temporal logics for information systems and object orientation we rely on has been developed in this setting (e.g. [Ser80; SFSE89; FSMS91; FM92; SCS94; SSC95; SSR96]).

However, certain "subtleties" of concurrent systems are lost under such simplifications (see, for instance, [Pra86] for a discussion on the subject). The need for a precise notion of causality as reflected by the time structures adopted for the logics led to the study of partial order temporal logics [PW84; KP87; Pen95]. The basic difference is that the assumption of an omnipresent observer of the entire system being considered is dropped and replaced by a local causal perspective. It is the case of the logics for partially ordered computations introduced by Pinter and Wolper [PW84], of interleaving set temporal logics [KP87] and of temporal logics for reasoning about distributed transition systems [LPRT95], systems with product state spaces [Thi95], occurrence nets [Rei92] and event structures [Pen88].

n-agent temporal logics can be found within the latter. They adopt event structures [Win87] enriched with information about its sequential agents [LT87] as models of concurrent systems. This approach already complies with the fact that the view each individual agent may have of the whole distributed system at a particular instant in time is partially due to the relative autonomy and spatial separation between agents, and has to be supported by communication [LRT92]. n-agent logics can explicitly distinguish sequential agents (localities) in the system, refer to the local viewpoint of each agent, and express communication between agents (the major feature of distribution) [LT87; LRT92; Ram96].

Several versions of n-agent logics can be found, still reflecting different perspectives on how non-local information can be accessed by each agent [Ram96]. The logics D_0 and D_1 we propose assume that an assertion about another agent is only possible at a communication point. But for instance, the logics in [LMRT91] assume that, at each instant, the actual information about another agent is the one corresponding to the last communication with it.

These have already been addressed in the context of object orientation [ES95] and used to axiomatize a significant subset of the GNOME language [Cal96].

Other n-agent logics [Ram96] do even consider the existence of a "present knowledge" modality allowing reference to non-local properties of agents, cf. [KR94].

For a detailed account on logics of knowledge see [FHMV95].

6.8 CONCLUDING REMARKS

Most of the alternative versions of distributed temporal logics referred to above have been found to have sound and complete axiomatizations. Also decidability results are available for most of them [LMRT91; LRT92; Pen95; Ram96]. Therefore, although no effort has been done yet in that direction, there is good hope for the development of a robust proof system for logics such as D_0 and D_1. As pointed out by Emerson in [Eme90], such "exogenous" logics with multiple time frames are specially well suited for compositional or modular specification and verification [BKP84]. This has already been identified by [Ram96] for several n-agent logics.

For the sake of simplicity, we adopt a very elementary system view in this chapter: just sets of objects operating concurrently and interacting synchronously. In real systems, objects will often be structured: they may be aggregated into complex objects, related by inheritance, or related in many other ways. For an effective specification method and its underlying logic, in-the-large mechanisms are indispensable for putting such structures together. [Ehr97] gives an approach how to treat these issues in a way compatible with the ideas developed in this chapter.

One of the most important object relationships is inheritance. Inheritance describes how a class reuses features from another one. On a specification level, the inheriting class adopts attributes and actions and possibly adds some. On an implementation level, the inheriting class reuses code, i.e., it implements inherited attributes and actions in the same way as in the original class. Most inheritance approaches allow for overriding of inherited actions: the name is kept but the implementation is replaced by a new one. It is not trivial to capture this in a logically clean and still manageable way.

Structuring may go further. What we have in mind is a module concept that reflects generic building blocks of software. In particular, parameterization and instantiation as well as horizontal and vertical composition of modules must be supported. For the latter, a module must incorporate a reification step between an external interface on a higher level of abstraction and an internal interface on a lower level of abstraction [DE95; Den96b; Den96a].

Other issues under investigation are real-time constraints and deductive capabilities. Real-time constraints are an important practical issue, a useful approach may be found in [DDPW94]. Deductive capabilities and default handling are harder to cope with but must eventually be better understood and incorporated.

Along with these theoretical developments, experimental languages, methods and systems have to be developed, supported by tools and tested in application case and field studies. The TROLL project has proceeded in this direction. A TROLL3/OMTROLL method has been designed that supports four views: system, object, behaviour and communication, along with advice how to proceed step by step. A TROLL3/ OMTROLL workbench is under construction where a set of tools is being designed for developing and validating TROLL3/OMTROLL specifications. Besides textual and graphical editors and their parsers, an animator is envisaged for validating specifications against user requirements, and to give support for exploring actual states by a variety of means. Finally, a TROLL3/OMTROLL application study is being performed for developing an information system within the German national institute for weights and measures, cf. section 6.6 above, and also [KKH+96; HDK+97].

Results and experiences are encouraging, but logics as discussed in this chapter may still take some time to find their way into practice. More research, both fundamental and experimental, is needed for understanding their intricacies and to find out how to incorporate them into a specification methodology.

Acknowledgments

We gratefully acknowledge inspirations from the editors, an unknown referee, members of our groups in Braunschweig and Lisbon, and Narciso Marti-Oliet.

References

[AB95] M. Aiguier and G. Bernot. Algebraic specification of object type specifications. In R.J. Wieringa and R.B. Feenstra, editors, *Information Systems—Correctness and Reusability, Selected Papers from the IS-CORE Workshop'94*, pages 16–32, 1995.

[AG91] A.J. Alencar and J.A. Goguen. OOZE: An object-oriented *Z* environment. In P. America, editor, *Proc. ECOOP'91*, pages 180–199. Springer, Berlin, 1991.

[AGM92] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum. *Handbook of Logic in Computer Science, Vols. 1-6*. Clarendon Press, Oxford, 1992.

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *8th ACM Symposium on Principles of Programming Languages, Williamsburg, VA*, pages 164–176, 1981. Also, *Acta Informatica*, 20(3):207-226, 1983.

[BK84] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109–137, 1984.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proc. 16th ACM Symp. Theory of Computing*, pages 51–63. ACM Press, 1984.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[BW90] J. Baeten and W. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, S. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object-oriented Development - The Fusion Method*. Prentice-Hall, 1994.

[Cal96] Caleiro, C. Distributed object communities. Master's thesis, Instituto Superior Técnico, Lisboa, 1996. *In Portuguese.*

[Cat94] R.G.G. Cattell. *Object Database Standard*. Morgan Kaufmann Publishers, San Mateo, 1994.

[CE81] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, pages 52–71. LNCS 131, 1981.

[CE98] C. Caleiro and H.-D. Ehrich. Temporal specification of distributed systems: The power of event calling. *Technical Report, Dept. de Matemática, Instituto Superior Técnico*, Lisbon 1998. *To appear.*

[Che76] P.P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *Communications of the ACM*, 1:9–36, 1976.

[DDPW94] E. Dubois, Ph. Du Bois, M. Petit, and S. Wu. ALBERT: A formal agent-oriented requirements language for distributed composite systems. In P. Hartel und G. Saake E. Dubois, editor, *Proc. Workshop on Formal Methods for Information System Dynamics (CAiSE'94)*, pages 25–39, 1994.

[DE95] G. Denker and H.-D. Ehrich. An Event-Based Semantics for Transactions. In G. Bernot and M. Aiguier, editors, *Proc. Int. Workshop on Information Systems – Correctness and Reusability (IS-CORE'95)*, pages 57–72. Universite d'Evry, LMI, Technical Report, 1995.

[Den96a] G. Denker. Semantic Refinement of Concurrent Object Systems Based on Serializability. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 105–126. Kluwer Academic Publ., 1996.

[Den96b] G. Denker. *Verfeinerung in objektorientierten Spezifikationen: Von Aktionen zu Transaktionen. Reihe DISDBIS*, Band 6. infix-Verlag, Sankt Augustin, 1996.

[EH83] E. Emerson and J. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *10th ACM Symposium on Principles of Programming Languages, Austin*, pages 127–140, 1983. Also, *Journal of the ACM*, 33(1):151-178, 1986.

[EH96] H.-D. Ehrich and P. Hartel. Temporal specification of information systems. In A. Pnueli and H. Lin, editors, *Proc. Int. Workshop in Honor of Chih-Sung Tang, World Scientific, Singapore*, pages 43–70, 1996.

[Ehr97] H.-D. Ehrich. Object Specification. In H.-J. Kreowski et al, editor, *IFIP 14.3 Volume on Foundations of System Specification, Chapter 11. To appear in Springer LNCS*, 1997.

[Eme90] E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 997–1072. Elsevier, 1990.

[ES95] H.-D. Ehrich and A. Sernadas. Local Specification of Distributed Families of Sequential Objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Types Specification, Proc. 10th ADT Workshop/5th COMPASS Workshop, Selected papers*, pages 219–235. Springer, Berlin, LNCS 906, 1995.

[Esp93] Espírito Santo Data Informática (ESDI), Lisbon. *OBLOG Users Manual*, 1993. Supplied with OBLOG-CASE v1.0 product kit.

[FHMV95] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about knowledge.* MIT Press, 1995.

[FL79] M. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *Journal of Computation and System Sciences*, 18:194–211, 1979.

[Flo67] R. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Mathematical Aspects of Computer Science - Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[FM92] J. Fiadeiro and T. Maibaum. Temporal theories as modularization units for concurrent systems specification. *Formal Aspects of Computing*, 4:239–272, 1992.

[Fra86] N. Francez. *Fairness.* Springer-Verlag, 1986.

[FS97] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, New York, 1997.

[FSMS91] J. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-theoretic semantics of object oriented constructs. In R. Meersman, W. Kent, and S. Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction, North-Holland*, pages 243–284, 1991.

[FW93] R.B. Feenstra and R. Wieringa. Lcm 3.0: A language for describing conceptual models – syntax definition. Technical report, Vrije Universiteit Amsterdam, 1993.

[GM87] J.A. Goguen and J. Meseguer. Unifying functional, object–oriented and relational programming with logical semantics. In P. Wegner B. Shriver, editor, *Research Direction in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[Gol92] R. Goldblatt. *Logics of Time and Computation*. CSLI, 1992.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, 1980.

[GS95] J.A. Goguen and A. Socorro. Module composition and system design for the object paradigm. *Journal of Object oriented Programming*, 7(14), 1995.

[GW88] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical report, SRI International, 1988.

[Har79] D. Harel. *First-Order Dynamic Logic*. Springer Verlag, LNCS 68, 1979.

[Har97] P. Hartel. *Konzeptionelle Modellierung von Informationssystemen als verteilte Objektsysteme*. Reihe DISDBIS. infix-Verlag, Sankt Augustin, 1997.

[HDK+97] P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. Information systems modelling with TROLL formal methods at work. *Information Systems*, 22(2-3):79–99, 1997.

[HJ95] P. Hartel and R. Jungclaus. Modeling Business Processes over Objects. *Int. Journal of Cooperative Information Systems*, 4(2):165–188, 1995.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[HM90] J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa85] C: A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.

[HSJ+94] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994.

[HZ92] J. Halpern and L. Zuck. A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, 1992.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering*. Addison-Wesley, Reading, MA, 1992.

[JSHS96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.

[JWH+94] R. Jungclaus, R. J. Wieringa, P. Hartel, G. Saake, and T. Hartmann. Combining TROLL with the Object Modeling Technique. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen. GI-Fachgespräch FG 1: Integration von semiformalen und formalen Methoden für die Spezifikation von Software*, pages 35–42. Informatik aktuell, Springer-Verlag, 1994.

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

[KP87] S. Katz and D. Peled. Interleaving set temporal logic. In *6th ACM Symposium on Principles of Distributed Computing*, pages 178–190. 1987.

[KR94] P. Krasucki and R. Ramanujam. Knowledge and the ordering of events in distributed systems. In *Proc. Theoretical Aspects of Reasoning About Knowledge*, pages 267–283. Morgan Kaufmann, 1994.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[LMRT91] K. Lodaya, M. Mukund, R. Ramanujam, and P.S. Thiagarajan. Models and logics for true concurrency. In P.S. Thiagarajan, editor, *Some Models and Logics for Concurrency, Advanced School on the Algebraic, Logical and Categorical Foundations of Concurrency. Gargnano del Garda*, 1991.

[LPRT95] K. Lodaya, R. Parikh, R. Ramanujam, and P. Thiagarajan. A logical study of distributed transition systems. *Information and Computation*, 119(1):91–118, 1995.

[LRT92] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Temporal logics for communicating sequential agents: I. *International Journal of Foundations of Computer Science*, 3:117–159, 1992.

[LS95] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148:303–324, 1995.

[LT87] K. Lodaya and P.S. Thiagarajan. A modal logic for a subclass of event structures. In Th. Ottmann, editor, *Proc. 14th Int. Colloq. on Automata, Languages and Programming*, pages 290–303. LNCS 267, Springer-Verlag, Berlin, 1987.

[Mes89] J. Meseguer. General Logic. In H.-D. Ebbinghaus et al, editor, *Logic Colloquium '87*, pages 275–329. North-Holland, Amsterdam, 1989.

[Mes92] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–156, 1992.

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.

[Mil80] R. Milner. *A calculus of communicating systems*. Springer Verlag, LNCS 92, 1980.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[Pel87] D. Peleg. Concurrent dynamic logic. *Journal of the ACM*, 34(2):450–479, 1987.

[Pen88] W. Penczek. A temporal logic for event structures. *Fundamenta Informaticae*, 11(3):297–326, 1988.

[Pen95] W. Penczek. Branching time and partial order in temporal logics. In L. Bolc and A. Szalas, editors, *Time and Logic: A Computational Approach*, pages 179–228. UCL Press, 1995.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 19th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society. New York, 1977.

[Pra86] V.R. Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[PW84] S. Pinter and P. Wolper. A temporal logic for reasoning about partially ordered computations. In *3rd ACM Symposium on Principles of Distributed Computing*, pages 28–37. 1984.

[Ram96] R. Ramanujam. Locally linear time temporal logic. In *Proc. 11th Annual IEEE Symposium on Logics in Computer Science*, pages 118–127. IEEE Computer Society. New York, 1996.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Rei92] W. Reisig. Elements of a temporal logic coping with concurrency. Technical report, Institut für Informatik, Technische Universität München, 1992.

[RS92] L. Rapanotti and A. Socorro. Introducing FOOPS. Technical report, PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, 1992.

[SCS94] A. Sernadas, J. F. Costa, and C. Sernadas. An institution of object behaviour. In H. Ehrig and F. Orejas, editors, *Recent trends in data type specification*, pages 337–350. Springer Verlag, LNCS 785, 1994.

[Ser80] A. Sernadas. Temporal aspects of logical procedure definition. *Information systems*, 5:167–187, 1980.

[SFSE89] 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 Systems Concepts: An In-Depth Analysis*, pages 225–246. North-Holland, 1989.

[SHJE94] G. Saake, T. Hartmann, R. Jungclaus, and H.-D. Ehrich. Object-oriented design of information systems: TROLL language features. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 1994.

[SJH93] G. Saake, R. Jungclaus, and T. Hartmann. Application Modelling in Heterogeneous Environments using an Object Specification Language. In M. Huhns, M.P. Papazoglou, and G. Schlageter, editors, *Int. Conf. on Intelligent & Cooperative Information Systems (ICICIS'93)*, pages 309–318. IEEE Computer Society Press, 1993.

[SR94] A. Sernadas and J. Ramos. The GNOME language: Syntax, semantics and calculus. Technical report, Tech. Report, Instituto Superior Técnico, Lisboa, 1994.

[SS77] J.M. Smith and D.C.P. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2:105–133, 1977.

[SSC95] A. Sernadas, C. Sernadas, and J.F. Costa. Object specification logic. *Journal of Logic and Computation*, 5:603–630, 1995.

[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, 1987. Morgan–Kaufmann, Palo Alto, 1987.

[SSR96] A. Sernadas, C. Sernadas, and J. Ramos. A temporal logic approach to object certification. *Data and Knowledge Engineering*, 19:267–294, 1996.

[Thi94] P.S. Thiagarajan. A Trace Based Extension of Linear Time Temporal Logic. In *Proc. 9th annual IEEE Symposium on Logic in Computer Science*, pages 438–447. IEEE Computer Society Press, 1994.

[Thi95] P. Thiagarajan. Ptl over product state spaces. Technical report, School of Mathematics, SPIC Science Foundation, Madras, India, 1995.

[vG90] R. van Glabeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Free University of Amsterdam, 1990.

[Wie96] R. Wieringa. *Requirements Engineering*. John Wiley & Sons, Chichester, 1996.

[Win87] G. Winskel. Event Structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, Proc. Advanced Course, Bad Honnef, September 1986*, pages 325–392. Springer, 1987. LNCS 255.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. *[AGM92]*, 4:1–148, 1995.

[Wol95] P. Wolper. On the relation of programs and computations to models of temporal logic. In L. Bolc and A. Szalas, editors, *Time and Logic: A Computational Approach*, pages 131–178. UCL Press, 1995.