LANGUAGE FEATURES FOR COOPERATION IN AN OBJECT-ORIENTED DATABASE ENVIRONMENT*

SUSAN J. EVEN, FRANS J. FAASE and ROLF A. DE BY

Faculteit Informatica, Universiteit Twente, P.O. Box 217
7500 AE Enschede, The Netherlands
E-mail: {seven, faase, deby}@cs.utwente.nl

Received 28 September 1995
Revised 3 May 1996

ABSTRACT

We introduce the language LOTOS/TM for the formal specification of a network of cooperating agents with a shared data repository and private local data. LOTOS/TM is the orthogonal integration of the process-algebraic protocol specification language LOTOS and the functional, object-oriented database specification language TM. The specified world consists of a number of interacting LOTOS processes—describing the cooperating agents—and a special LOTOS process representing the shared data repository, which is modeled as a TM database. The data repository's functionality is made available to the other, cooperating processes through one or more external database gates. Interaction at such a gate corresponds to a method invocation in the database. In addition to shared persistent data, the TM language is used to specify the data encapsulated locally within processes, and the transient data communicated over gates.

Some features of LOTOS/TM are inherently suitable for describing cooperation, such as combinators for synchronization on specific methods. These features are illustrated by examples showing navigation events on a shared graph structure that resembles a hypertext. Emphasis in the examples is placed on coordination aspects of the scenario. LOTOS/TM serves as a formalism for a more user-friendly specification language by the name of CoCoA that is currently under construction.

Keywords: LOTOS/TM, cooperation, coordination, synchronized database updates, navigation events

1. Introduction

The design of advanced computer systems that help to organize the work for some communally undertaken task is an area of research that has gained increased attention over the last decade. The data management problem has been identified by a number of authors, including [1, 2, 3]. Important reasons for the upsurge in interest more recently are the increased availability of fine-grain network technology

*This research was supported by the ESPRIT LTR Project TRANSCOOP (EP8012). TRANSCOOP is funded by the Commission of the European Communities. The partners in the TRANSCOOP project are GMD (Germany), Universiteit Twente (The Netherlands), and VTT (Finland).

at-the-fingertips on the one hand, and the understanding that formerly sequentially organized tasks can sometimes be performed more efficiently in parallel as a rule of thumb on the other hand. This only holds *provided* that the new task is well-designed. Attention, therefore, has also been paid to general questions of activity control, for instance by [4, 5]. Furthermore, some work has been done on the transaction support for CSCW by [6, 7] and others. A final dimension that has been studied is that of users and groups of users [8, 9]. The field, however, is fairly dynamic and consensus about generically applicable operational models has not yet been reached. It is generally believed that the existing, rich patterns of cooperation are insufficiently accommodated by existing technology and levels of control.

Typical characteristics of the field are, we believe, simultaneous activities by end-users in the cooperation, shared data on which the activities are focussed, and timely submissions of results of isolated activities. The key ingredients, thus, for a precise description of cooperatively undertaken tasks are, in our opinion, (1) *user processes*, (2) *shared data*, and (3) *cooperative transactions*. From a language design point of view, there is thus a need to define synchronous and asynchronous processes, data, and functions on the data.

One should observe that the design of *cooperative scenarios*—the name we use for the cooperatively undertaken tasks—is non-trivial: the scenarios involve inherently complex protocols dealing with a network of users, and are thus difficult to fully understand. It is our goal to formally describe these scenarios in such a way that we are capable of determining whether they satisfy certain characteristics such as lack of potential deadlock, the commutativity of certain user actions, or requirements on the allowed ordering of database method invocations.

Many of the description languages that are in use in industry or have been proposed in the literature are either incomplete in that they do not accommodate the definition of processes, data or data manipulation functions, or they seem to lack the formal foundation needed to answer the verification questions one may have regarding the scenarios. Our goal is to arrive at a well-founded and complete specification language. With this goal in mind, we explore the orthogonal integration of two very different, but already existing, formal specification languages: TM, an object-oriented database specification language [10, 11, 12], and LOTOS, a process-algebraic specification language for distributed systems [13, 14, 15]. Both languages have been in satisfactory use in their respective application areas for a number of years.

Features provided by the proposed LOTOS/TM language combination specific to supporting cooperation include facilities to specify shared, persistent data using an object-oriented data model, and facilities to define complex synchronization and communication protocols among user processes which access the shared data. This general framework is characteristic of workflow systems, cooperative document authoring, and design for manufacturing, three application areas we are studying in the context of the ESPRIT TRANSCOOP Project (8012).[a]

[a] TRANSCOOP documents are available at: `http://wwwtranscoop.cs.utwente.nl:8080/`.

A LOTOS/TM specification consists of a LOTOS part and a TM part. Together, the LOTOS and TM parts define the behavior and the shared and communicated data of a cooperative scenario. The LOTOS part identifies a network of user processes and an additional, special process that manages the shared database, which is specified in TM. The database is made available to the user processes through one or more external database gates. (A process gate, in the LOTOS sense, is a port through which the process can communicate with its environment in a predefined way. When such an act of communication occurs, we call it an event at the gate.) An event at a database gate corresponds to a method invocation in the database, and this constitutes (one part of) the link between the two languages. The other part is achieved by also defining the data local to a user process, and the data communicated between user processes in TM.

Accesses to the database can be specified in a cooperative way. For example, a number of user processes may synchronize on a particular method invocation: each participating process may provide arguments to the method invocation, and the processes negotiate on the values of the arguments. We refer to these synchronized operations as *cooperative updates*. This form of cooperation is expressed simply and elegantly in LOTOS/TM by synchronization combinators, which are parameterized by the database gates. The specification designer may choose the granularity at which cooperation is to take place by the way methods are associated with gates: separate update and retrieval gates may be used to synchronize update operations and interleave retrievals (a feature useful for describing shared window applications), or the cooperation required for each method may be defined independently by using a gate for each method.

In addition to describing shared persistent data, the functional nature of its methods allows TM to be used for declaring and manipulating local data within LOTOS processes. A process may encapsulate local data and define how it can be accessed by other processes by offering events upon which the other processes can synchronize. This facility is useful for specifying private workspace data, for which the sharing of partial results can be controlled by the encapsulating process.

An important advantage for simulation purposes comes from the observation that an appropriate transaction model for CSCW ought to allow the concurrent usage of the same data items by multiple users. The LOTOS/TM set-up uses external gates to access the database, and we can use the language to actually define, through a LOTOS behavior expression, how these external gates are provided. We thereby define the behavior of the database (i.e., the allowed ordering of method invocation events at the gates), and thus simulate its transaction model. This behavior expression serves as an interface to the database, and can be defined independently from the processes that access the database. An added benefit is thus that we can experiment with a number of transaction models, and empirically identify which transaction model features are most wanting.

We stress that the LOTOS/TM language combination is being proposed mainly as a complete and well-founded formalism for CSCW applications. It has not been

designed with the goal of user-friendliness in mind. To that purpose, we are currently working on a design language that is formally founded on LOTOS/TM, yet is easier to use, in part because of a graphical interface. The preliminary design of this end-user language, called CoCoA, is reported in [16].

This paper is organized as follows. In Section 2, we provide summary descriptions of the core languages LOTOS and TM and discuss their integration as far as LOTOS declarations and value expressions go. The issue of language combination is taken further in Section 3, where we outline the use of TM value expressions within LOTOS processes. We also study the (im)possibility of modeling data persistency in LOTOS in that section. As we have already identified shared, persistent data as one of the key ingredients for CSCW, Section 4 introduces external database gates as a way to access and manipulate persistent data. Section 5 gives examples to illustrate cooperation in the language, and proposes additional features that take advantage of TM's data type facilities. Section 6 makes comparisons to related work. Finally, Section 7 discusses open issues in the design and implementation of the language and gives an overview of ongoing and future work.

2. Structure of LOTOS/TM

A LOTOS/TM specification consists of a TM part and a LOTOS part. The TM part specifies a shared database; the LOTOS part specifies a number of processes that access the database. We assume a single TM database for now; Section 7.1 discusses the extension to multiple databases. The TM language is introduced in Section 2.1; Sections 2.2 and 2.3 describe LOTOS.

2.1. TM

TM is an object-oriented (and type-theoretic) data model that is theoretically well-founded, and capable of expressing many, if not all, features of data models currently in use [10, 11, 12]. It is formally founded in a typed lambda calculus allowing for subtyping and multiple inheritance, based on the ideas of [17]. Characteristic features of TM are the distinction between types, classes, and sorts; support for object identity and complex objects; and multiple inheritance of data structure, methods and constraints.

A TM specification defines a database by its typed attributes and a set of methods that can be invoked on the database. Types of attributes may be arbitrarily complex: the type constructors supported are tuple ($\langle \cdots \rangle$), variant ($[\![\cdots]\!]$), set ($\mathbb{P} \ldots$), and list ($\mathbb{L} \ldots$). An abstract syntax for TM types is given in Figure 1. A specification is organized in terms of sort and class definitions. Sorts define complex value types; sort definitions include methods (which are applied like functions, taking a sort value as an argument), and constraints that the values must satisfy. Classes define object instance types; class definitions include object methods and constraints (for a single object instance), plus class methods and constraints (for collections of object instances). Specialization between classes (and sorts) is de-

d : PrimitiveType
τ : TypeExpression
S : SortOrClassName (user-defined)
ℓ : Label

d ::= **int** | **real** | **bool** | **char** | **string** | **oid** | **error** | **nil**
τ ::= d | S | $\langle \ell_1 : \tau_1, \ldots, \ell_k : \tau_k \rangle$ | $[|\ell_1 : \tau_1, \ldots, \ell_k : \tau_k|]$ | $\mathbb{P}\tau$ | $\mathbb{L}\tau$

Fig. 1. An abstract syntax for TM types.

noted by the ISA-clause, which may identify several superclasses. A class inherits the attributes, constraints, and methods of its superclasses.

Together with a collection of sort and class definitions comes a database definition, which closely resembles a class definition. In the database definition, a list of persistent variables is presented together with their type. (A common type for such a variable is $\mathbb{P}C$, where C is some defined class.) These variables constitute the database state. This is achieved formally by viewing the variables as attribute labels of a record, which is the database object. Besides persistent variable declarations, the database definition also includes constraints that the database state has to obey, and definitions of methods that operate on the database state.

Database methods have an implicit **self** argument, which stands for the database object as a whole. Methods come in two flavors: *retrieval methods* and *update methods*. Retrieval methods (or queries) take the database and retrieve some information from it. Update methods take the database and change its state. Both forms of method may require parameters. The language for method definition is a full-fledged functional language that incorporates simple arithmetic and first-order boolean operators, as well as an extensive range of set operators for database queries in the style of complex object SQL [18]. This is opposed to the common practice in industry where methods are directly defined in procedural languages like C++, which we believe is an obvious (and unfortunate) obstacle to verifiable software code. A constraint analysis tool has been developed for the verification of TM methods [19].

2.2. Basic LOTOS

LOTOS (Language Of Temporal Ordering Specifications) is a language that is designed for the formal specification of distributed, concurrent systems [13, 14, 15]. LOTOS is based on process-algebraic ideas. The core of the language, so-called Basic LOTOS, is formed by a pure process algebra. (Full LOTOS is Basic LOTOS extended with guards, value expressions and variable declarations—see Sections 2.3 and 3.)

A LOTOS process definition is a hierarchical structure identifying subprocesses, their synchronization and data communication. A process is viewed as a 'black-box computing agent' capable of performing unobservable, internal actions, and interactions with its environment via so-called gates that it is said to offer. Such

interactions are called *events*. (Sub)processes are defined in terms of behavior expressions; behavior expressions bottom out in event names (or gates).

Possible forms of behavior expressions B are given in Figure 2, which we have adopted from [13]. The table shows so-called Basic LOTOS behavior expressions.

expression name	syntax B ::=
inaction	**stop**
action prefix	
– internal	$\mathbf{i}$; B
– observable	$\mathbf{g}$; B
choice	B_1 [] B_2
process instantiation	$\mathbf{p}[\mathbf{g}_1, \ldots, \mathbf{g}_n]$
successful termination	**exit**
parallel composition	
– general case	B_1 $\|[\mathbf{g}_1, \ldots, \mathbf{g}_n]\|$ B_2
– pure interleaving	B_1 $\|\|\|$ B_2
– full synchronization	B_1 $\|\|$ B_2
hiding	**hide** $\mathbf{g}_1, \ldots, \mathbf{g}_n$ **in** B
enabling	$B_1 \gg B_2$
disabling	B_1 [> B_2

Fig. 2. Basic LOTOS behavior expression syntax.

The **stop** process is the process incapable of any (inter)action. Action prefix allows to express that an action occurs before some behavior. The general case parallel composition identifies n gates on which the behaviors B_1 and B_2 have to synchronize. Pure interleaving is a special case, namely where $n = 0$. Likewise, full synchronization is the case where the set of gates $\{ \mathbf{g}_1, \ldots, \mathbf{g}_n \}$ is the intersection of the sets of gates offered by B_1 and B_2. Hiding allows to turn actions at the mentioned gates into internal actions of the process. Behavior expressions can be parameterized by gate names to obtain process definitions. Process instantiation assigns actual gate names to the gate parameters. The **exit** action is the only way to successfully terminate a process. Enabling is like action prefix: it allows to express ordering of events, but is meant for general behavior expressions. Disabling, finally, allows to express that a normal behavior B_1 may, at any moment, be disrupted by another behavior B_2.

The dynamic semantics of LOTOS is defined by a transition system. Labels in the transition system correspond to actions, both internal and observable as gate events. Rules describe how behavior expressions are rewritten as actions occur. Execution of a LOTOS specification is analogous to derivation in the transition system. A good introduction to the operational semantics of LOTOS is found in [13]. A LOTOS tool set environment for simulation, compilation and prototype generation is described in [20].

2.3. Full LOTOS with TM

Full LOTOS introduces value communication and data types to the process-algebraic concepts of Basic LOTOS. The original data language for Full LOTOS is based on the ACT-ONE theory of abstract data types [21]. For LOTOS/TM, we use TM as the data language instead. The constructs of Full LOTOS that involve value expressions are briefly introduced below.

In Full LOTOS, an event (or gate) name may be followed by value and variable declarations, together also called *event attributes*. An event name together with its attributes is referred to as a *structured event* or *event offer*. A value declaration looks like $!e$, where e is an allowed value expression in the data language. A variable declaration takes the form $?x{:}\,\tau$, where x is a variable identifier and τ is a data type. An example of a LOTOS structured event is the expression:

$$\mathtt{g}\ !e_1\ ?x{:}\,\tau\ [e_2]\ ,$$

where g is a gate name, e_1 is a value expression, $x{:}\,\tau$ is a variable declaration, and e_2 is an expression of type `bool`, called a *selection predicate*. Value expressions and variable declarations may occur intermixed. The order in which they occur is important for synchronization. The selection predicate is used to constrain possible values for synchronization. For example, the following two event offers:

$$\mathtt{g}\ !3\ ?x{:}\,\mathbf{int}\ [x > 0] \qquad \text{and} \qquad \mathtt{g}\ ?y{:}\,\mathbf{int}\ !5\ [y > 0]$$

can synchronize, since the gate names match, and the proposed attribute values are correctly typed and satisfy the selection predicates. Synchronization means that the event g !3 !5 happens. The above event offers could be made by different processes, $\mathtt{p}_1$ and $\mathtt{p}_2$, composed with a combinator for synchronization on gate g as follows: $\mathtt{p}_1$ |[g]| $\mathtt{p}_2$.

There are other possible occurrences of value expressions in Full LOTOS that are not part of structured events. These include actual parameters to process instantiations other than gate names, guards as conditions on possible behavior, and the **let** construct for local value definitions. Section 3 discusses these constructs in more detail.

3. Local process data

The syntax of LOTOS/TM allows the use of TM expressions within LOTOS process definitions. LOTOS variables are declared to have TM types and TM values are bound to these variables. We will refer to variables declared in a LOTOS process definition as the *local data* of the process. (A mechanism to manipulate *global data* will be introduced in Section 4.) Every TM type that is mentioned in the *interface* of the database (i.e., as a parameter or result type of a database method) can be used in the LOTOS definitions. Additional TM types can be defined for value encapsulation, manipulation, and communication by the processes to supplement those in the database interface. In this paper, only TM sort types will be used

in the LOTOS part of a specification. The manipulation of objects outside of the database is discussed in Section 7.2.

We start off this section with a discussion of variables and persistency in LOTOS. Next, we introduce the language constructs of LOTOS/TM that involve value expressions and variable declarations. The remaining sections discuss expression evaluation and the binding of TM values to LOTOS variables.

3.1. Variables and persistency

It is important to understand both the nature of local process data, and the limitations of its use. In LOTOS, there is no notion of "variable" in the traditional, imperative sense of the word. A variable names a value, not a storage location. Variable declarations are local to a process definition, and it is not possible to declare variables that are shared by more than one process. This has important consequences on the use of LOTOS to describe and manipulate persistent and shared data.

Persistency is specified in LOTOS using process parameters and recursion: to be retained, a value must be passed on as a parameter to a recursive instantiation of a process. Sharing of persistent data is modeled either by a complex value-passing protocol or by an interface process (for an encapsulated value) with which other processes synchronize. The data that is to be shared must be passed on as a parameter to all processes that share it, or the shared data must be enclosed within a process definition that offers operations on the data as synchronization events (with the parameters and results of these operations communicated as event attributes). An example of this type of modeling is given in Section 5.1. This modeling technique is, however, limited, since persistency and sharing must be specified by the designer.

Section 4 introduces external database gates as a means for LOTOS processes to access and manipulate shared, persistent data. The remainder of this section discusses the use of TM for local process data.

3.2. Value expressions and variable declarations

In LOTOS/TM, LOTOS process variables are declared to have TM types. LOTOS/TM expressions (TM expressions that may contain LOTOS variables) will provide the values for these variables. Figure 3 gives a syntax for LOTOS/TM expressions. Observe that expressions can be built using LOTOS variable identifiers. The syntax does not refer to items in the database schema other than methods: there is no way to refer directly to database attributes, except via the methods that access the attributes. The syntax domain 'ValueIdentifier' is used for local value definitions *inside* of expressions (e.g., as in the **collect** and **where** constructs).

Figure 4 lists the LOTOS/TM constructs that contain value expressions. The constructs are understood as follows. The **let** construct introduces typed abbreviations for value expressions within a behavior expression. A process instantiation supplies actual gate and value parameters to the process. A process returns results

e : Expression
I : ValueIdentifier
x : LotosIdentifier
m : MethodIdentifier
ℓ : Label
τ : TypeExpression

$$
\begin{aligned}
e \ ::= \ & \mathrm{I} \mid x \mid m[e](e_1, \ldots, e_n) \mid e \cdot \ell \\
& \mid [|\ell = e|] \mid \langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle \mid \{e_1, \ldots, e_n\} \\
& \mid \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif} \mid e \textbf{ except } (\ell_1 = e_1, \ldots, \ell_k = e_k) \\
& \mid \textbf{replace } e_1 \textbf{ for } \mathrm{I} \textbf{ in } e_2 \textbf{ iff } e \mid e \textbf{ as } \tau \mid e_1 \textbf{ isa } e_2 \\
& \mid \textbf{collect } e_1 \textbf{ for } \mathrm{I} \textbf{ in } e_2 \textbf{ iff } e \mid \textbf{emptyset}(\tau) \\
& \mid e_1 \textbf{ subset } e_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \\
& \mid \textbf{forall } e_1 \textbf{ in } e_2 \textbf{ | } e \mid \textbf{exists } e_1 \textbf{ in } e_2 \textbf{ | } e \\
& \mid \textbf{not } e \mid e_1 \textbf{ or } e_2 \mid e_1 \textbf{ and } e_2 \mid e_1 \textbf{ implies } e_2 \mid e_1 \textbf{ equiv } e_2 \\
& \mid e_1 \textbf{ in } e_2 \mid e \textbf{ is } \ell \mid e \textbf{ on } \ell \mid (e) \\
& \mid \textbf{case } e \textbf{ of } \ell_i = \mathrm{I}_i : e_i \textbf{ endcase} \\
& \mid e_1 \textbf{ union } e_2 \mid e_1 \textbf{ intersect } e_2 \mid e_1 \textbf{ minus } e_2 \\
& \mid \{\mathrm{I} \textbf{ in } e_1 \textbf{ | } e_2\} \mid e \textbf{ where } (\mathrm{I}_1 = e_1, \ldots, \mathrm{I}_n = e_n) \\
& \mid \textbf{unique for } \mathrm{I} \textbf{ in } e_1 \textbf{ iff } e_2 \mid \cdots
\end{aligned}
$$

Fig. 3. An abstract syntax for LOTOS/TM expressions. This syntax only represents a subset of TM, but it is sufficient for the examples we present later.

construct	syntax B ::=
LET bindings	**let** $x_1 : \tau_1 = e_1, \ldots, x_n : \tau_n = e_n$ **in** B
process instantiation	$\mathbf{p}[\mathbf{g}_1, \ldots, \mathbf{g}_n](e_1, \ldots, e_m)$
exit parameters	**exit**$(e_1, \ldots, e_k)$
value declarations and selection predicate of an event offer	$\mathbf{g} \cdots !e_i \cdots [e]$
guards	$[e] \rightarrow$ B

Fig. 4. Value expressions in LOTOS/TM.

by parameterizing an **exit** action with value expressions. A guard is a condition (a boolean TM expression) that must be fulfilled to make a behavior possible; if the guard is not fulfilled, the guarded behavior expression is equivalent to **stop**.

Figure 5 lists the LOTOS/TM constructs that declare variables. Process definitions can be recursive: they can be specified as non-terminating (**noexit**), or they can terminate with or without values (indicated by **exit** parameters). The **accept** construct is used to accept the results of a behavior expression. The construct is used in conjunction with process enabling as shown. B_1 must terminate with an action of the form **exit**$(e_1, \ldots, e_k)$. In contrast to function application, LOTOS processes that return values cannot be instantiated and used as value expressions, because the instantiation of a process definition is a behavior, not a value.

construct	syntax B ::=
LET bindings	**let** $x_1 : \tau_1 = e_1, \ldots, x_n : \tau_n = e_n$ **in** B
process parameters	**process** p $[\mathbf{g}_1, \ldots, \mathbf{g}_n]$ $(x_1 : \tau_1, \ldots, x_m : \tau_m)$: F := B **endproc** F can be **noexit**, **exit**, or **exit**$(\tau_{m+1}, \ldots, \tau_{m+k})$; it is called the *functionality* of behavior B.
accept parameters	$B_1 \gg$ **accept** $x_1 : \tau_1, \ldots, x_k : \tau_k$ **in** B_2
variable declarations of an event offer	$\mathbf{g} \cdots ?x_i : \tau_i \cdots$

Fig. 5. Variable declarations in LOTOS/TM.

3.3. Expression evaluation without side effects

Expressions in the ACT-ONE data language of Full LOTOS are defined and simplified using equational specification techniques. The semantics guarantees that value expressions are evaluated without side effects. (A LOTOS variable is bound to a value only once, and hence always refers to the same thing.) In order to integrate LOTOS and TM in an orthogonal way, we have guaranteed that LOTOS/TM expressions are also evaluated without side effects. Section 2.1 introduced TM as a functional language, but it should be obvious to the reader that database update methods will change the state of the database. Thus, one would expect some TM expressions to have side effects. Let's take a closer look at the forms of method invocation that can occur within LOTOS/TM expressions.

The syntax of method invocations in Figure 3 shows the functional application of a TM method. It takes the form $m[e](e_1, \ldots, e_n)$, where the **self** argument, e, is supplied as an explicit parameter, between square brackets. When a TM update method is applied in a functional way, it returns the modified value as a result. The result of an update can only be made permanent if the updated value is kept (e.g., used instead of the previous value). Closer inspection of the constructs of

LOTOS/TM reveals that it is not possible to update the database from within LOTOS/TM when methods are applied like functions. There are two reasons for this. First, in order to apply a database update method within a LOTOS/TM expression, the database state needs to be supplied as an explicit **self** parameter. But how can we get a copy of the database state to use as this value? This is not possible. Second, in order to apply a database update method in such a way that it changes the database state, the result of applying the method must be kept. But, assuming we can obtain a new value for the database state, how can we commit this value? Given the language features in Figures 3, 4 and 5, there is no way to do this. The updated value can be bound to a LOTOS variable, but it cannot be made persistent. How to evaluate a database update method as a state change is addressed in Section 4.

Within LOTOS/TM expressions, method invocations are used to manipulate *local* process data. Because the method applications are purely functional, the expressions containing their invocations are without side effects. Earlier, we restricted the TM data types that can be used within LOTOS/TM processes to be TM sorts. Objects can also be manipulated in a functional way as local process data, provided we define the semantics of an object in an appropriate way. Section 7.2 discusses issues related to object retrieval and update. The next section discusses binding TM values to LOTOS variables.

3.4. Binding TM values to LOTOS variables

When should LOTOS/TM expressions be evaluated? Full LOTOS uses a combination of syntactic substitution and equational simplification for its ACT-ONE data types. Syntactic substitution is used for **let** bindings and process instantiations. Equational simplification is used when terms must be compared (e.g., during synchronization). The ADT equations are used to simplify ground terms (terms without free identifiers) in order to determine what transition rules are applicable to a behavior. Processes that are to synchronize must apply transitions labeled with the same label (i.e., the same gate name and the same event attributes). Terms (value expressions) are thus simplified (evaluated) as part of state transitions. Any problems with simplification (e.g., due to incompleteness of the equation system) will mean that an event "doesn't happen."

For the evaluation of LOTOS/TM expressions, we will use a combination of syntactic substitution and function application. Syntactic substitution will be used for binding TM expressions to LOTOS variables in **let** constructs and process instantiations. Because TM method invocations within LOTOS/TM expressions are purely functional, syntactic substitution does not cause problems for expression evaluation, since the value of an expression does not depend on an external state. For the remaining binding mechanisms (**accept** bindings and variable declarations in event offers), evaluation of LOTOS/TM expressions to values will be done as part of an event. If there is a problem with expression evaluation during a synchronization attempt (e.g., due to a TM constraint violation), the event will not happen.

As an example, we describe the synchronization of two event offers. Variables declared in the event offers are bound to values as the result of successful synchronization. For example, synchronization of the structured events g $!e_1$ $?\mathtt{x}_2{:}\,\tau_2$ and g $?\mathtt{x}_1{:}\,\tau_1$ $!e_2$, where e_1 has type τ_1 and e_2 has type τ_2, results in the binding of the value of e_1 to $\mathtt{x}_1$ and the value of e_2 to $\mathtt{x}_2$. LOTOS identifiers occurring in e_1 must be bound by the environment of the first event; the scope of $\mathtt{x}_2$ does not include e_1. The same holds for the LOTOS identifiers of e_2 and the scope of $\mathtt{x}_1$ in the second event. Evaluations of e_1 and e_2 are without side effects.

In addition to the binding mechanisms listed in Figures 4 and 5, TM values can also be bound to LOTOS variables by means of database retrieval methods. This is discussed in Section 4.

4. External database gates

Section 3.1 pointed out that shared storage in LOTOS cannot be external to a process definition. This has obvious disadvantages when it comes to specifying a database. A LOTOS/TM specification will therefore define a TM database that is *external* to all LOTOS process definitions. Collectively, we refer to the process definitions as the *scenario* specification.

An interface for the database will offer method invocations as events. These events take place at *database gates*. Methods are invoked by LOTOS processes through synchronization with the database interface at these gates. The **self** parameter to a method invocation event is implicit and is the database state. The database gate mechanism thus hides the database state from the scenario processes. Evaluation of a method invocation will thus depend on when the event happens with respect to the rest of the scenario specification.

4.1. Declarations and persistency

The TM **module** construct is used to declare the persistent attributes of the database object. Modules methods define the operations that will be offered at one or more database gates. For the impatient reader wanting to see code, Figures 7 and 9 give examples of TM sort and module specifications. The attribute `DIAGRAM` declared in Figure 9 is persistent; the methods `links`, `groups` and `traverse` defined in Figure 9 can be offered by one or more database gates associated with the module.

A database is made available to the LOTOS part of the specification as an external gate parameter as follows:

specification `scenario` [`db`] $\cdots$:= $\cdots$ **endspec**

where `db` stands for the name of a TM database (i.e., **module** definition), and the keyword `scenario` introduces the LOTOS part of the LOTOS/TM specification.

A database gate can be split into two distinct gates, one for update method invocations and one for retrieval method invocations, if desired. Because LOTOS does not have true parallelism—parallel composition has an interleaving semantics—splitting the database gate does not pose problems: methods cannot be invoked at

both gates simultaneously; their invocations are interleaved. We will denote distinct update and retrieval gates by $\mathtt{db_U}$ and $\mathtt{db_R}$, respectively:

$$\textbf{specification}\ \mathtt{scenario}\ [\mathtt{db_U}, \mathtt{db_R}]\ \cdots\ :=\ \cdots\ \textbf{endspec}$$

This indicates a partitioning of the update and retrieval methods (defined in **module** `db`) among the two gates. The distinction is useful for describing cooperation dependent on the kind of method. (An example is given in Section 5.3.)

The partitioning of methods among gates can be used to control the granularity at which processes cooperate. Because attributes and methods are uniquely named in TM, we may use one gate per database method as follows:

$$\textbf{specification}\ \mathtt{scenario}\ [\mathtt{m}_1, \ldots, \mathtt{m}_k]\ \cdots\ :=\ \cdots\ \textbf{endspec}$$

where $\mathtt{m}_1$ to $\mathtt{m}_k$ are method names. In this way, the synchronization for each method can be specified independently.

Processes synchronizing on a database gate must agree on what methods are invoked, and negotiate on what arguments are used for each invocation. A database gate can be seen as a mechanism for grouping together those database operations that will be synchronized upon in a similar way because of cooperation or negotiation requirements on parameters to the operations. Before discussing the form method invocation events will take, we first consider the testing of database constraints.

4.2. Constraint testing

Although TM allows the description of sort, object, class, and database constraints, it does not address testing them. *When* constraints are tested is considered to be an implementation matter by the designers of TM; the issue of transactional boundaries is unaddressed. Because LOTOS provides a context for TM method invocations, arguments to a method can sometimes be tested for applicability before the method is invoked, thus offering a means to avoid constraint violations or undefined results due to improper arguments. But this is not always possible, and LOTOS/TM must deal with the possibility of constraint violation when a method is invoked. Failures of methods invoked through the database gate will be dealt with differently than failures of methods applied (functionally) to local data. Methods applied to local data are tied to expression evaluation and the occurrence of events (if a method invocation fails, the event does not happen); whereas methods invoked through the database gate will return status information about the success of the operation.

4.3. Method invocation events

The form of method invocation events at a database gate will vary slightly depending on how methods are associated with the gates. As mentioned above, three different associations are possible: a single database gate, separate update

and retrieval gates, or one gate for each method. In the first and second cases, the gate name must be parameterized with the name of the method to be invoked; in the third case, the method name is the gate name. An advantage of the first two associations is that synchronization combinators are easier to express, since they do not need to include long lists of method names. Our examples will use separate update and retrieval gates.

Updates Update events at a database gate will take the form:

$$\mathtt{db_U}\ !\mathit{method\_name}\ !\mathit{arg}_1 \cdots !\mathit{arg}_n\ !v$$

where arg_1 to arg_n are input arguments to the method, and v is a value of type **bool** indicating the success of the method invocation[b]. The types of the input arguments arg_1 to arg_n must match the signature of the method definition. Observe that each argument in the method event schema above is written as a value (!) declaration. The schema represents that of the *event* that occurs, not the *event offers* of the LOTOS processes involved in the event.

For synchronization with the database, values must be provided for all method arguments. However, these values might be provided by different processes. For example, consider the event offers:

$$\mathtt{db_U}\ !m\ !s_1\ ?w\colon \mathbf{string}\ ?x\colon \mathbf{bool} \qquad \text{and} \qquad \mathtt{db_U}\ !m\ ?z\colon \mathbf{string}\ !s_2\ ?y\colon \mathbf{bool}$$

where s_1 and s_2 are strings, m is the name of an update method offered at gate $\mathtt{db_U}$ that takes a pair of strings as arguments, and x and y are variables to hold status information. These event offers may synchronize together with the database, resulting in the occurrence of a *successful* method invocation event: $\mathtt{db_U}\ !m\ !s_1\ !s_2\ !\mathbf{true}$, or an *unsuccessful* method invocation event: $\mathtt{db_U}\ !m\ !s_1\ !s_2\ !\mathbf{false}$. Notice that the first event offer provides the first argument s_1 to the event, and the second event offer provides the second argument s_2 to the event. We refer to such multi-participant database events as *cooperative updates*. Section 5.3 discusses the idea further and gives an example.

The specification designer is free to use either a single update method event offer, with a variable declaration for the success value:

$$\begin{array}{l}\mathtt{db_U}\ !m\ !s_1\ ?w\colon \mathbf{string}\ ?\mathtt{success}\colon \mathbf{bool}\ ;\\ (\quad [\mathtt{success}] \rightarrow \cdots\\ \quad\ \ [] \\ \quad [\mathbf{not}\ \mathtt{success}] \rightarrow \cdots\ )\end{array}$$

or a pair of event offers, as follows:

$$(\mathtt{db_U}\ !m\ !s_1\ ?w\colon \mathbf{string}\ !\mathbf{true}\ ;\ \cdots)$$

[b] We choose to use a boolean value here, rather than introduce a special type of value for **success** and **failure**. We believe this design choice is justified by our desire not to store such values in the database, as would be suggested by making them part of the type system.

$$\Big[\Big]$$
$$(\mathtt{db_U}\ !m\ !s_1\ ?w{:}\,\mathbf{string}\ !\mathbf{false}\ ;\ \cdots)$$

The first event offer can only happen if the update method invocation completes successfully, without violating any constraints. The second event offer can only happen if the update is *not* successful (e.g., it violates constraints). In order for an update *event* to occur, the process offering the above events must synchronize with the database, as well as with another process that provides the second argument to the method. This argument value will be bound to w as a result of the synchronization. If another process does not offer a value for w, synchronization with the database is not possible, so neither event occurs. If the method invocation (or constraint testing as a result of the invocation) never terminates, neither event happens.[c]

Retrievals Retrieval events at a database gate will take the form:

$$\mathtt{db_R}\ !\mathit{method\_name}\ !\mathit{arg}_1 \cdots !\mathit{arg}_n\ !v$$

where v is a value of the variant type $[|\mathtt{okay}{:}\,\tau, \mathtt{error}{:}\,\mathbf{error}|]$, and τ is the result type of the method.[d] For retrieval methods, we do not have the option to use an additional event attribute to indicate its return status. If a retrieval method invocation fails, its result value (also an event attribute) is undefined.

As for update method invocations, the above schema represents a retrieval method invocation event that occurs at a database gate. Thus, all attributes to the event are written as value declarations. Again, values must be provided for all method arguments, but these values might be provided by different LOTOS processes.

A LOTOS process may synchronize with a retrieval event using the following syntax to test the variant result of the method invocation:

$$\begin{array}{l}\mathtt{db_R}\ !m\ !e_1 \cdots !e_n\ ?x : [|\mathtt{okay}{:}\,\tau, \mathtt{error}{:}\,\mathbf{error}|]\ ; \\ (\quad [x\ \mathbf{is}\ \mathtt{okay}] \rightarrow\ \mathbf{let}\ \mathtt{result}{:}\,\tau = (x\ \mathbf{on}\ \mathtt{okay})\ \mathbf{in}\ \cdots \\ \qquad \Big[\Big] \\ \quad\ [x\ \mathbf{is}\ \mathtt{error}] \rightarrow \cdots\ )\end{array}$$

The process offering these events can synchronize with the database, without the help of another process, if it provides values for all of the arguments to the method. Observe that removal of the `okay` tag from the result is somewhat cumbersome.

---

[c] If more than one method invocation event is possible (i.e., different methods can be invoked at the database), non-terminating evaluation should not cause other method invocations to be blocked. This is an important concern for the LOTOS/TM simulation environment.

[d] Type **error** has a unique value: the constant **error**. For clarity of expression, we also use `error` as a label.

Syntactic sugaring would allow the specifier to use a pair of events, as follows:

$$
\begin{array}{l}
(\mathtt{db_R}\ !m\ !e_1 \cdots !e_n\ ?\mathtt{result}{:}\,\tau\ ;\ \cdots) \\
\quad [] \\
(\mathtt{db_R}\ !m\ !e_1 \cdots !e_n\ !\mathbf{error}\ ;\ \cdots)
\end{array}
$$

The first event automatically removes the `okay` tag from the variant result, and assigns the value to a variable of type τ. Our implementation of LOTOS/TM allows this simplified syntax when it does not cause ambiguities.

4.4. Method composition

We will use the term *method composition* to refer to the construction of multi-step database operations using existing method definitions. We can provide method composition in LOTOS/TM *statically*, by means of the database specification, or *dynamically*, by means of the behavior specification. These options are explained below.

Static composition TM methods are atomic operations that we can define at a granularity to suit the application. The definitions of several methods can be composed and offered as another method in the database schema. Such a method will be made atomic by the specification language. This solution is not always feasible, however, since applications often require the possibility to dynamically compose methods.

Dynamic composition In order to specify multi-step database operations in LOTOS, we need language constructs for glueing together method invocation events. Events can be sequenced using the action prefix combinator (;). The following code sequences the two method invocation events:

```
  db_U !m_1 !e_1 !true
; db_U !m_2 !e_2 !true
  ...
```

Unfortunately, the method invocation events of one process might be interleaved with those of another process (depending on the rest of the specification), so simply specifying that method invocations are sequenced does not create a multi-step database operation.

The LOTOS/TM specification and/or the database interface process (see Section 4.5) can restrict the interleaving of method invocations. The database interface process can be defined to offer locking operations or *start-transaction*, *end-transaction* operations to a LOTOS/TM specification. Examples that use locking operations to control interleaving can be found in [22]. The operations for supporting method composition will necessarily depend on the transaction model used.

4.5. The database interface

A database interface process can be defined (by the system designer) to specify the behavior at the external database gates. The interface process offers method invocations as events at the gates and defines the allowed orderings of the invocations. The interface process is placed in parallel composition with the behavior part of a LOTOS/TM specification. As above, we assume two database gates: $\mathtt{db_U}$ and $\mathtt{db_R}$. The synchronization combinator $|[\mathtt{db_U}, \mathtt{db_R}]|$ is used as follows to compose an interface process with a scenario process:

$$\mathtt{interface}[\mathtt{db_U}, \mathtt{db_R}] \ \ |[\mathtt{db_U}, \mathtt{db_R}]| \ \ \mathtt{scenario}[\mathtt{db_U}, \mathtt{db_R}]$$

The scenario process defines the cooperative application that uses the database. The two processes synchronize on all of the events that occur at gates $\mathtt{db_U}$ and $\mathtt{db_R}$. The interface process must not only offer method invocations as events, it must also supervise the interleaving of the invocations. For example, the interface might check whether method invocations are ordered to preserve database constraints. Additional operations (events) can be offered by the interface process to extend the database functionality to provide the features of an advanced transaction model. For the TRANSCOOP project we plan to simulate the cooperative transaction model described in [23].

Method invocation events are made by all processes that share a database, yet TM does not address multi-user services. To extend TM with multi-user, transactional capabilities, locking operations and lock acquisition protocols for a database can be provided by its interface process. An example of an interface process is given in Figure 6. This process offers transactional access to an encapsulated TM

```
process interface [dbU, dbR, open, abort, commit] (self : TM_value) : noexit :=
    open ?tid : int ;
    interfaceT [dbU, dbR, open, abort, commit] (self, tid)
    ≫ accept commit : bool, newself : TM_value in
          interface [dbU, dbR, open, abort, commit]
                    (if commit then newself else self endif)
endproc
where process interfaceT [dbU, dbR, open, abort, commit]
                         (self : TM_value, tid : int) : exit(bool, TM_value) :=
         abort !tid ; exit(false, self)
       [] commit !tid ; exit(true, self)
       [] ... (operations on self) ...
      endproc
```

Fig. 6. Example of a simple transaction rollback facility.

value. A transaction is opened with a transaction identifier (the `tid` value); a new subprocess is instantiated for the transaction, and subsequent operations use the `tid` as a parameter to ensure the abort or commit operation is made by the same transaction.

5. Specifying cooperation in LOTOS/TM

In this section, we provide some illustrative examples of the suitability of the LOTOS/TM combination to specify cooperation. The examples address navigation through a shared graph structure. Nodes and links between nodes are specified. The data existing at each of these nodes is assumed to be independent. Each user is associated with a particular node and issues operations on the data at that node in cooperation with the other users at that node. Users may traverse the links connecting the nodes in order to switch to a different node, and hence cooperate with a different group of users.

Emphasis is placed on coordination aspects of the scenario. We do not address the manipulation of data at the nodes, only the maintenance of the graph and its traversal. The offering of traversal events is controlled by the specification. Notification messages are sent to all users whenever a user switches nodes; this provides group awareness.

We develop the example in two stages: first, local process data is used to maintain the graph (see the *d* parameter of the `navigator` process in Figure 8); then the graph is moved to an external database (see the **module** definition in Figure 9), and the methods are refined to resemble complex queries. Our presentation is intended to illustrate the features of LOTOS/TM and includes only specification fragments.

5.1. Navigation through an encapsulated structure

This section gives an example of a LOTOS/TM specification of a shared graph structure. The TM part of the specification (see Figure 7) describes the shared diagram, constraints it must satisfy, and operations on its structure. The LOTOS part of the specification (see Figure 8) offers graph traversal operations as events. The graph structure and traversal events are intended to dynamically guide the cooperations that take place among the different users active in the diagram. The combined LOTOS/TM specification:

- maintains the shared graph structure: a LOTOS/TM process encapsulates the diagram; users synchronize with this process to access the diagram.
- maintains activity group nodes: each node dynamically defines an activity group, based on what users are currently associated with the node.
- enables graph traversal according to the structure: participants can dynamically enter and exit node groups by traversing links.
- sends automatic notification when a user changes groups: the `notify` event is offered to each user; it is parameterized with the user's current group and the set of user names associated with that group.

We assume the set of users associated with a particular node group to be cooperating. The participants in this kind of cooperation are dynamically determined.

```
Sort Diagram
    type ⟨START : ⟨node_name : string⟩,
          NODES : ℙ ⟨node_name : string⟩,
          LINKS : ℙ ⟨link_name : string, from_node : string, to_node : string⟩,
          USERS : ℙ ⟨user_name : string, user_group : string⟩⟩
    constraints
        startokay : START in NODES
        keynode : NODES key node_name
        keylink : LINKS key link_name
        keyuid : USERS key user_name
        usersokay : forall u in USERS |
            (exists n in NODES | (n.node_name = u.user_group))
        linksokay : forall n in LINKS |
            ((n.from_node ≠ n.to_node)
             and (exists n1 in NODES | (n1.node_name = n.from_node))
             and (exists n2 in NODES | (n2.node_name = n.to_node)))
    retrieval methods
        link_defined(in link : string, out bool) =
            exists t in LINKS | (t.link_name = link)
        node_defined(in grp : string, out bool) =
            exists t in NODES | (t.node_name = grp)
        user_defined(in usr : string, out bool) =
            exists t in USERS | (t.user_name = usr)
        can_traverse(in usr : string, link : string, out bool) =
            if not link_defined[self](link) then false
            else exists t in USERS | (t.user_name = usr and t.user_group = grp)
                 where (grp = from[self](link)) endif
        who(in grp : string, out ℙ string) =
            collect u.user_name for u in USERS iff (u.user_group = grp)
        group(in usr : string, out string) =
            (unique for u in USERS iff (u.user_name = usr)).user_group
        from(in link : string, out string) =
            (unique for t in LINKS iff (t.link_name = link)).from_node
        to(in link : string, out string) =
            (unique for t in LINKS iff (t.link_name = link)).to_node
    update methods
        migrate(in usr : string, link : string) =
            if not can_traverse[self](usr, link) then self
            else self except (USERS = replace ⟨user_name = usr,
                                               user_group = to[self](link)⟩
                                      for u in USERS iff (u.user_name = usr))
            endif
end Diagram
```

Fig. 7. TM sort specification of a shared graph structure.

```
process navigator [traverse, notify] (d : Diagram) : noexit :=
    traverse ?usr : string ?link : string [can_traverse[d](usr, link)]
    ; let dd : Diagram = migrate[d](usr, link) in
      ( ( par u : ⟨user_name : string, user_group : string⟩ in dd.USERS
            ||| (notify !u.user_name !u.user_group !who[dd](u.user_group) ; exit)
        ) ≫ navigator [traverse, notify] (dd)
      )
endproc (* navigator *)
```

Fig. 8. LOTOS/TM specification of navigation events through the shared graph structure.

Process `navigator` offers graph traversal operations as events at gate `traverse`. Synchronization at this gate is possible when appropriate user and link arguments are provided. After a traversal event occurs, a notification event is offered at gate `notify` for each user, giving the members in the user's group. Subsequent traversal events are offered by the recursive instantiation of the process using the modified (post-traversal) graph structure.

The `navigator` process highlights the use of TM expressions as event attributes. It also illustrates the testing of method arguments for constraint satisfaction before applying a method. For example, the `traverse` event has the selection predicate:

$$[\ \texttt{can\_traverse[d](usr, link)}\ ]$$

which is identical to the test in the update method `migrate`, declared in sort `Diagram`.

We can use this example to take a closer look at value generation in the context of a selection predicate, and what it entails for LOTOS/TM. The semantics of LOTOS provides for an anonymous value to be bound to those variables declared in an event which are not bound to a value when the event occurs. For example, LOTOS would allow the event:

$$\texttt{traverse}\ \texttt{?usr:}\mathbf{string}\ \texttt{?link:}\mathbf{string}\ [\ \texttt{can\_traverse[d](usr, link)}\ ]$$

to synchronize with the event: `traverse` ?u: **string** ?n: **string**, for some anonymous pair of strings, s_1 and s_2, for which `can_traverse[d]`(s_1, s_2) holds. Future computations can be dependent on the values chosen. At present, we do not know how to treat value generation in LOTOS/TM (there is no way to introduce anonymous "constants" of an arbitrary TM type). A practical approach will likely be taken for the LOTOS/TM simulation environment by requesting the user to provide appropriate values.

The example introduces the use of a TM-specific LOTOS construct: generalized parallelism over a finite set. We use the syntax: **par** $x : \tau$ **in** e ||| B_x, where e is a TM expression of type $\mathbb{P}\tau$ and B_x is a behavior expression dependent on x. The construct allows the specification of *and parallelism*. In Figure 8, the behavior

expression:

par u : ⟨user_name : **string**, user_group : **string**⟩ **in** dd.USERS
||| (notify !u.user_name !u.user_group !who[dd](u.user_group) ; **exit**)

specifies the finite interleaving of one parallel behavior for each element u_i of dd.USERS:

(notify !u_1.user_name !u_1.user_group !who[dd](u_1.user_group) ; **exit**)
||| ⋯
||| (notify !u_k.user_name !u_k.user_group !who[dd](u_k.user_group) ; **exit**)

If the set dd.USERS is empty, the construct rewrites to the **exit** behavior. All of the interleaving behaviors must complete before the composite behavior can terminate.

Obviously, the semantics of the **par** construct is dependent on the collection being finite.[e] Moreover, in order to know the value of dd.USERS, the construct requires migrate[d](usr, link) to be evaluated (syntactic substitution is used for the binding of dd), but this evaluation is not done as part of an event. (The notify events must happen for *all* of the users in dd.USERS, and this cannot be expressed as a selection predicate.) We plan to investigate what forms the interleaved behavior expressions can take, and whether to allow other parallel combinators to be used. Although mapping the construct to an implementation is only possible when the **par** collection is finite, we feel the usefulness of the construct to describe cooperation outweighs its problems.

5.2. Navigation events at a database gate

We now move the encapsulated data structure of the previous section to an external module definition, making it an attribute of the database. Figure 9 declares a persistent attribute of sort Diagram in module N. New methods are defined to query the diagram and retrieve tuples of information. Module methods are invoked by the navigator process in Figure 10. The retrieval methods links and groups are invoked through database gate $\mathtt{N_R}$; the update method traverse is invoked through database gate $\mathtt{N_U}$. Observe that the same traversal and notification events are offered by the processes in Figures 8 and 10. Both offer a choice of traversal events, one for each valid user-link combination. For the latter process, traversal events also synchronize with the database. Queries are done to retrieve information regarding the offering of traversal and notification events.

The example is suggestive of things we would like to be able to prove about our specification: for example, the fact that user_name is a key for USERS (see Figure 7) implies that all users are sent a notification message.

[e] A prototype safeness detector for TM has been developed that can be used to statically test some forms of TM expression for finiteness [24].

```
module N
(* definition of sort Diagram...  *)
module section
    attributes
        DIAGRAM: Diagram
    module retrieval methods
        links(out ℙ ⟨usr : string, links : ℙ string⟩) =
            collect ⟨usr = u.user_name,
                    links = collect n.link_name for n in self.DIAGRAM.LINKS
                            iff (n.from_node = u.user_group)⟩
            for u in self.DIAGRAM.USERS
        groups(out ℙ ⟨name : string, group : string, members : ℙ string⟩) =
            collect ⟨name = u.user_name,
                    group = u.user_group,
                    members = who[self.DIAGRAM](u.user_name)⟩
            for u in self.DIAGRAM.USERS
    module update methods
        traverse(in usr : string, link : string) =
            self except (DIAGRAM = migrate[self.DIAGRAM](usr, link))
end N
```

Fig. 9. TM module specification of the shared graph structure.

```
process navigator [N_R, N_U, notify]  :  noexit :=
    N_R !links ?pairs : ℙ ⟨usr : string, links : ℙ string⟩ ;
    ( choice p : ⟨usr : string, links : ℙ string⟩ in pairs
        [] (N_U !traverse !p.usr ?n : string !true [n in p.links] ; exit) )
    ≫
    N_R !groups ?result : ℙ ⟨usr : string, group : string, members : ℙ string⟩ ;
    ( par t : ⟨usr : string, group : string, members : ℙ string⟩ in result
        ||| (notify !t.usr !t.group !t.members ; exit) )
    ≫ navigator [N_R, N_U, notify]
endproc (* navigator *)
```

Fig. 10. LOTOS/TM specification of navigation events with database gates.

The example introduces a second TM-specific LOTOS construct: generalized choice over a set of values.[f] The following syntax:

$$\begin{array}{l}\textbf{choice } p : \langle \texttt{usr} : \textbf{string}, \texttt{links} : \mathbb{P}\ \textbf{string}\rangle \textbf{ in } \texttt{pairs} \\ \quad [] \ (\ \texttt{N}_\texttt{U}\ !\texttt{traverse}\ !p.\texttt{usr}\ ?n : \textbf{string}\ !\textbf{true}\ [n \textbf{ in } p.\texttt{links}]\ ;\ \textbf{exit}\ )\end{array}$$

specifies the offering of one behavior for each element p_i of `pairs`:

$$\begin{array}{l}\quad (\ \texttt{N}_\texttt{U}\ !\texttt{traverse}\ !p_1.\texttt{usr}\ ?n : \textbf{string}\ !\textbf{true}\ [n \textbf{ in } p_1.\texttt{links}]\ ;\ \textbf{exit}\ ) \\ [] \ \cdots \\ [] \ (\ \texttt{N}_\texttt{U}\ !\texttt{traverse}\ !p_k.\texttt{usr}\ ?n : \textbf{string}\ !\textbf{true}\ [n \textbf{ in } p_k.\texttt{links}]\ ;\ \textbf{exit}\ )\end{array}$$

The choices offered above can also be expressed without the **choice** construct by means of a complex selection predicate on the database event:

$$\begin{array}{l}\texttt{N}_\texttt{U}\ !\texttt{traverse}\ ?u : \textbf{string}\ ?n : \textbf{string}\ !\textbf{true} \\ \qquad [\textbf{exists } p \textbf{ in } \texttt{pairs} \mid p.\texttt{usr} = u \textbf{ and } n \textbf{ in } p.\texttt{links}]\end{array}$$

But the generalized choice construct seems to express the possible traversal events more intuitively than does the selection predicate.

5.3. Cooperative updates

LOTOS/TM can be used to specify cooperative updates in which multiple users synchronize on a method invocation event at an external database gate. Parameters to an invocation are provided in a cooperative manner by the different users. As an example, consider a system in which two users perform tasks independently. Suppose all updates to a shared database should be agreed upon by both participants. We indicate this below by the combinator $|[\texttt{db}_\texttt{U}]|$, which expresses synchronization on a database update gate:

$$\texttt{user}[\texttt{db}_\texttt{R}, \texttt{db}_\texttt{U}](name_1)\ |[\texttt{db}_\texttt{U}]|\ \texttt{user}[\texttt{db}_\texttt{R}, \texttt{db}_\texttt{U}](name_2)$$

Cooperative update operations wait for shared commitment to the database. A cooperative database update does not happen until all participants are ready to perform it. The specification indicates that retrievals at the database gate $\texttt{db}_\texttt{R}$ can be interleaved. For this example, it does not matter in what order the shared data is read, as long as the updates are synchronized upon. The distinction of two gates, $\texttt{db}_\texttt{U}$ and $\texttt{db}_\texttt{R}$, corresponds to a partitioning of the methods by functionality. We might further distinguish the granularity at which cooperation is to take place by specifying one gate for each method.

We saw an example of a cooperative update event in Section 5.2. The `navigator` process in Figure 10 can synchronize with user processes on gates $\texttt{N}_\texttt{U}$ and `notify`

[f] This construct is more specific than the generalized choice over a sort provided by Full LOTOS. In contrast to the **par** construct introduced earlier, the selection set need not be finite.

as follows:

$$\begin{array}{l}\texttt{navigator}[\texttt{N}_\mathbf{R}, \texttt{N}_\mathbf{U}, \texttt{notify}] \\ \quad |[\texttt{N}_\mathbf{U}, \texttt{notify}]| \\ (\texttt{user}[\texttt{N}_\mathbf{R}, \texttt{N}_\mathbf{U}, \texttt{notify}](name_1) \ |[\texttt{N}_\mathbf{U}]| \ \texttt{user}[\texttt{N}_\mathbf{R}, \texttt{N}_\mathbf{U}, \texttt{notify}](name_2))\end{array}$$

All $\texttt{N}_\mathbf{U}$!`traverse` events will be synchronized upon by the processes, as indicated by the combinators; the $\texttt{N}_\mathbf{R}$!`links` and $\texttt{N}_\mathbf{R}$!`groups` events of process `navigator` are done without cooperation, as are any retrievals done by the user processes. Notification events offered by the `navigator` process synchronize independently with each user process, with the user's name given as an event attribute.

6. Related work

Given the nature of LOTOS/TM as the integration of two already existing languages, and given the fact that the integration is orthogonal, with the only modification to either language being the use of Full LOTOS with TM values, Section 6.1 starts off by comparing our work to current proposals for extensions to LOTOS. Section 6.2 gives commentary on cooperative document authoring, an application area we are working to describe in LOTOS/TM as part of the TransCoop project. We focus on one research effort in particular that uses LOTOS to describe a synchronous editing environment. Section 6.3 discusses previous and related work on integrating LOTOS and TM. This last section is included for completeness.

6.1. Extensions to LOTOS

E-LOTOS is an international research effort to develop an ISO standard for Extended LOTOS. The project was established in 1993 to identify and propose enhancements for LOTOS. The proposals include (but are not limited to) an improved data language, typed gates, time quantification of events, non-synchronized termination, exception handling, and suspend and resume capabilities. For obvious reasons, we would like LOTOS/TM to be compatible with E-LOTOS. [25] is a working document that summarizes the proposed extensions. The 1995 E-LOTOS meetings have resolved to replace the ACT-ONE data type language of Full LOTOS with a two-level language based on an ML-like functional approach [26] and an equational (ACT-ONE or extended ML) approach. [27] details the proposed data type extensions. The new data language will be formally defined, with a strongly typed, strict functional semantics. The following features are proposed: built-in type constructors such as records, variants, lists and arrays (with functional update); recursively-defined (first order) functions (and hence non-termination); subtyping (particularly on records); and support for exception handling. It is important for the data type extensions to be compatible with existing LOTOS specifications, and therefore support is needed for the equational definition of data types. It is not yet clear what the relationship between the functional and equational data type languages for E-LOTOS will be.

TM meets most of the above requirements, the exceptions being array and function types, equational data type definition, exception handling, and a strict semantics. Although arrays and functions are not available as built-in types in TM, the language does provide ample type definition facilities to describe them. TM provides a rich collection of built-in data types that includes sets. Constraints can be defined to further specify the legal values of a type. Equational type definition is not supported in TM.

Exceptions would be a useful extension to the TM language, as they would introduce a convenient way to handle failures arising from database constraint violations. As mentioned in Section 4.2, TM does not address the failure of expression evaluation because it is a specification language. The downside of introducing exceptions is that they would make TM implementational in some respects. Exception handling is a data type issue that also affects the definition of LOTOS. A syntax for exception handling within LOTOS processes would be needed, and the dynamic semantics of LOTOS would need revision, since exceptions raised during data evaluation could give rise to behavior transitions. We are not in a position to resolve this issue for LOTOS/TM; we can only await the outcome of the E-LOTOS standard.

Using a strict semantics for TM would not always be desirable. TM allows the definition of predicated (and hence infinite) sets. This could be an obstacle to meeting the strictness requirement, since the evaluation of an expression that describes an infinite set may not terminate. Such an expression is said to be *unsafe*. Unsafe expressions can affect the semantics of LOTOS. For example, consider trying to synchronize two processes on an infinite value: value comparison would not terminate. Non-termination and failure of constraints were the reasons for tying expression evaluations to the occurrence of events in Sections 3.3 and 4.3. A proof tool exists for checking some forms of TM expression for safeness [24]. With the help of such a tool, the use of potentially unsafe forms of TM expression could be restricted within LOTOS, if necessary. This will need to be investigated when the E-LOTOS standard is available.

6.2. Cooperative document authoring

Cooperative document authoring is one of the application areas we are studying in the TRANSCOOP project. Related work on cooperative authoring typically falls into two categories: models derived from an implemented system that lack a formal description (e.g., [28] and [29, 30, 31, 32]), and formal models that are limited to operations on simple data types, such as inserting or deleting a character or drawing a brush stroke (e.g., [33] and [34]). The former approaches define a conceptual model, based on a system that has been successfully implemented; the latter approaches use simplistic data models. In contrast, our work emphasizes formal specification, and uses a powerful, object-oriented data model. Below, we discuss one of these approaches, which uses LOTOS to describe cooperative authoring.

LOTOS specification of a CSCW tool [34] investigates the use of Full LOTOS to specify the CSCW tool We-Met (Window Environment Meeting Enhancement Tools) [35]. We-Met provides a drawing area in which several users can work simultaneously; it is based on synchronous communication. The LOTOS specification distinguishes actions initiated by user displays (each display is modeled by a process) and those broadcast by the central We-Met system (also modeled as a process). All displays synchronize upon the actions broadcast by the central system. As in our work, LOTOS synchronization combinators are used to describe the coordination required for shared views. The examples are confined to a simple stack data structure, however, and only the signatures are given for the data operations. Our work, on the other hand, focuses on the cooperative use of more complex forms of shared data, as found in an object-oriented database.

6.3. Integration of LOTOS and TM

Database interoperability [36] discusses the use of LOTOS and TM to specify protocols for database interoperability. The languages are used in orthogonal ways to formally describe different aspects: LOTOS is used to specify the interconnection protocols for cooperation; TM sort definitions are used to specify the communicated data types. Examples are of a travel agency with local branches and a distributed seat reservation system. LOTOS behavior expressions are given to define the ordering of database operations at each site, and to define the interactions between sites. The examples are restricted to the Basic LOTOS subset of Full LOTOS, and only method names are used in the LOTOS processes, without method parameters. The use of TM values within LOTOS is not addressed. LOTOS/TM is more expressive than the language of [36] in that it allows the specification of synchronizations that depend on arguments to method invocations.

Merge options for LOTOS and TM [22] analyses three different approaches for an orthogonal merge of LOTOS and TM: the LTM-value approach (TM values are encapsulated in a version of Full LOTOS), the LTM-gate approach (persistent TM data is made available to LOTOS processes through external database gates), and the LTM-event approach (a LOTOS scenario event may synchronize with a number of TM method invocations at different databases). The first two approaches form the basis of the work we have presented here; the third approach is described below. Examples are given to motivate each merge option. Different specification styles are used to emphasize different aspects of cooperation: a task-oriented specification style for steps and sequencing, an agent-oriented specification style for communication between participants, and a data-oriented specification style for the interleaving of operations on shared data. The reader is referred to [22] for practical examples of the use of LOTOS and TM to describe workflow, mailing agents, locking protocols, and commutativity tables.

LTM events [37] identifies a high-level primitive for cooperative scenario definition, called the LTM event. These are LOTOS/TM events in which a number of method invocations at different databases are associated with the occurrence of a single event in a network. An LTM event has built-in transactional capabilities, and is, in fact, an abstraction of detailed protocols that take place underneath, between the different databases involved in the event. In such an event, local database updates are dependent on the results of queries against remote databases. An LTM event offer is comprised of a LOTOS event offer, and optional *pre* and *post* update methods. The *pre* update method sets up local workspace involved in the event. Execution of an LTM event imposes dependencies on value communication and synchronization. The *post* update method is evaluated after the LOTOS event. All required input values to the *post* update must be obtained as a result of queries on the local database, or as values communicated during synchronization with remote databases. LTM events are designed to express the synchronization of method invocations at a number of databases—something that is not possible in LOTOS/TM. However, the protocols needed to implement LTM events might be specified in LOTOS/TM, thereby providing the same features at a lower level.

An integrated semantic model [38] and [39] look at the integration of LOTOS and TM in terms of their semantical models. The formal semantics of LOTOS is defined as a transition system [15], whereas the formal semantics of TM is denotational [11]. If a transition system semantics is designed for TM, it becomes possible to define an integrated semantic model for the combined language that is a product of the two semantic models (transition systems). Emphasis is on the events and state transitions that take place; the syntactical representation of TM values within LOTOS processes (i.e., the forms of TM expression that appear in event offers, etc.) is not addressed. This work is ongoing.

7. Conclusions and future work

LOTOS/TM integrates the LOTOS and TM specification languages in two complementary ways: TM methods are applied in a functional way to local data encapsulated by a LOTOS process; TM methods are applied in an imperative way to shared external data by means of database gates and method invocation events. Both possibilities can be used to describe cooperation among processes that share data. Encapsulated local data can be used when fine-grained protocols are needed to specify how sharing is to take place. Access and manipulation of local data is specified explicitly. Method invocation events on external data can be used both for synchronization of independent computations and for negotiated database operations. Synchronized database update events express cooperation simply and elegantly, using a single language construct. The remainder of this section gives an overview of our ongoing and future work on LOTOS/TM.

Section 7.1 discusses the application of LOTOS/TM to multiple databases. Section 7.2 discusses two open issues in the design of LOTOS/TM and outlines their

partial solution. Section 7.3 discusses limitations of the language with regard to declarative, high-level specification and user-friendliness. Section 7.4 outlines our plans for an integrated tool set environment for LOTOS/TM, built on the existing tool sets for the two languages.

7.1. Extension to multiple databases

The structure of a LOTOS/TM specification can be extended to accommodate multiple databases. In this case, the TM part specifies a number of independent databases, each defined using the **module** construct. Each LOTOS process is parameterized by external gates to the database modules it uses. As for the case of a single database, every TM type that is mentioned in the interface of a database module (as a parameter or result type of a method) can be used in the LOTOS part of the specification. Additional types can be defined for value encapsulation, manipulation, and communication by LOTOS processes. Processes that communicate data must share the types of the communicated data, otherwise synchronization is not possible.

Database modules as objects In TM, objects and databases both have records as their underlying type. All attributes of an object, and all attributes declared in a database are persistent. The only difference between a database and an object is the lack of an **oid** for a database; it is not possible to refer to the attributes of a database other than by name. Nevertheless, TM databases are objects in the Oblique sense [40]. Oblique is a language that supports distributed object-oriented computation, with network references to objects; its objects are persistent records (without **oid**s) that are local to a site and referred to by name. In our extension of LOTOS/TM to multiple databases, each **module** definition will declare a database; the module's methods will define the operations that can be applied to the database. It is possible to declare TM modules on a scale small enough to be considered "normal" (i.e., `Person`-type) objects, and thus very similar to Oblique objects in that they are referred to by name, rather than **oid**. When external gates are associated with such small-scale "database modules as objects," a fine-grained specification of the database interface becomes possible: one can specify the allowed interleaving of method invocations at these gates, and the events at these gates affect only a small amount of persistent data.

The database interface The TM methods invocable at a database are atomic; they serve as primitive building blocks for the construction of cooperative scenarios. In the extension to multiple databases, we require that the methods of one database do not directly invoke the methods of other databases. Instead, composite operations that involve multiple databases need to be defined in the LOTOS part of a specification. Thus, the processes that share the databases are responsible for organizing the desired cooperation.

An interface that offers method invocations as events can be specified for the case

of multiple databases by combining the behaviors defined for all of the independent databases. The behaviors of different databases can be interleaved by composing their interface processes with the ||| combinator:

$$(\texttt{interface}_{\texttt{M}}[\texttt{M}](\cdots) \;|||\; \texttt{interface}_{\texttt{N}}[\texttt{N}](\cdots)) \;|[\texttt{M},\texttt{N}]|\; \texttt{scenario}[\texttt{M},\texttt{N}](\cdots)$$

The combinator emphasizes the fact that all of the databases function independently.

The usual issues in distributed database management need to be tackled when cooperation involves several databases. Because the interfaces offer method invocations as events, the composition of the interfaces must ensure that the ordering (interleaving) of method invocations at different database gates does not cause problems. For example, deadlock could be possible when more than one database is locked during a multi-database operation. To solve these problems, a more complex interface to the databases could be defined to offer transactional operations over multiple databases, perhaps using additional gates.

7.2. Open issues

There are two open issues in the design of LOTOS/TM: the evaluation of selection predicates with value generation, and the retrieval and update of database objects. These are discussed below.

Value generation with selection predicates In Full LOTOS with ADTs, it is not necessary for all variables to be bound to values when a selection predicate is evaluated; some variables acquire values by way of the ADT equations. Consider the selection predicate of the following LOTOS/TM event:[9]

$$\texttt{notify}\ !u.\texttt{user\_name}\ !u.\texttt{user\_group}\ ?g : \mathbb{P}\,\textbf{string}\ [g = \texttt{who}[\texttt{dd}](u.\texttt{user\_group})]$$

Depending on the synchronization attempt, the selection predicate might need to be evaluated without a binding for variable g (i.e., when synchronization with an unconstrained ?-declaration is attempted). However, to evaluate the selection predicate as a TM expression, g must be bound, since both arguments to the "=" operator are evaluated to values before the test for equality. Full LOTOS value generation would *assign* the result of the method invocation to variable g, but this conflicts with the semantics of expression evaluation in TM. The evaluation of selection predicates in combination with value generation requires further investigation, but simple equivalences like the one above that involve a ground TM expression on one side, and a LOTOS variable on the other side might be allowed, since value generation in this case is trivial.

[9] This event is equivalent to the event specified in Figure 8; the method invocation has been moved to the selection predicate.

Object retrieval and update Earlier, we restricted database interface types to be TM sorts. Retrieving objects from a database introduces a number of important questions. Consider binding a retrieved object to a process variable. What the retrieved object denotes is important when we consider applying methods to the object. It is not sufficient to retrieve an **oid** from the database. In order to apply retrieval methods to the object (e.g., in a local workspace), the values of its attributes must also be retrieved. Similarly, if some of its attributes refer to other objects, then the values of their attributes also need to be retrieved, and so on. Our implementation of LOTOS/TM will need to rely on a versioning mechanism to support object retrieval, since the values of a retrieved object's attributes should not be affected by database updates that happen after the object was retrieved.

Updating a retrieved object complicates things further. Recall that LOTOS/TM expression evaluation is without side effects, and methods are applied in a functional way to process data. Because of this, only *copies* of objects can be retrieved from the database (be bound to process variables) and be manipulated as local process data, since applying update methods to a retrieved object will not modify the database. The local modifications made to an object copy must be explicitly saved to the database by invoking update methods at the database gates.

7.3. Declarative, high-level specification

As a result of experimentation with the proposed LOTOS/TM language, we are aware that specifications to describe even simple workflow scenarios can quickly become unmanageable in size. In some respects, the LOTOS/TM language seems too "low level" to be used by scenario designers. It is necessary for the designer to describe some aspects of the cooperation that might be automatically provided by the system. For this reason, more declarative constructs for cooperation will be introduced as "macro" extensions to the LOTOS/TM language. We have chosen to call this extended language CoCoA (Coordinated Cooperation Agendas). The CoCoA language extensions will be mapped to LOTOS/TM for formalization. The more declarative constructs of CoCoA will aid in the definition of the essential elements of cooperation, such as users, user workspaces, and protocols for sharing resources and partial or intermediate results. The work on CoCoA is ongoing [16].

7.4. Implementation of a tool set

Both LOTOS and TM have tool sets to aid specification design. We plan to integrate existing tool sets for the two languages to allow users to take advantage of the tools already available, in addition to the new tools we develop for the combined language. The LITE tool set for LOTOS includes a graphical browser, a simulation environment, and two compilers for prototype generation [20]. The TM tool set includes a graphical interface, a type checker, and a prototype generator [41].

Our implementation of LOTOS/TM will provide a simulation environment for visualizing the cooperative behavior of a scenario specification. The simulator will

allow the designer to visualize the offering of method invocation events, and to "walk through" a specification. The simulator will be designed along the lines of the Smile simulation environment for LOTOS [42, 43]. The step-by-step unfolding of a behavior will offer choices of possible events to the user for selection, including operations at the database gates, and process synchronization and communication events that do not involve the database. Because event offers include LOTOS/TM expressions, event selection will initiate an expression evaluator for their step-by-step evaluation.

Another essential component of the tool set will be a type checker. Many specification errors involve data types and can be statically detected. The usage of LOTOS/TM gate names that correspond to database gates will be checked against the method signatures in the database schema for correct typing. An overview of the specification environment we are developing for LOTOS/TM and CoCoA is found in [44].

References

1. I. Greif and S. Sarin, "Data sharing in group work," *ACM Transactions on Office Information Systems* **5**, April 1987, 187–211.
2. A. H. Skarra, "Concurrency control for cooperating transactions in an object-oriented database," *Proc. ACM SIGPLAN Workshop on Object-based Concurrent Programming*, San Diego, CA, September 1988, pp. 145–147.
3. S. Greenberg (ed.), *Computer-supported Cooperative Work and Groupware*, Computers and People Series, Academic Press Ltd., 1991.
4. K. Kuutti, "The concept of activity as a basic unit of analysis for CSCW research," *Proc. Second European Conference on CSCW*, Amsterdam, September 1991, L. Bannon, M. Robinson and K. Schmidt (eds.), pp. 249–264.
5. T. Rodden and G. Blair, "CSCW and Distributed Systems: The Problem of Control," *Proc. Second European Conference on CSCW*, Amsterdam, September 1991, L. Bannon, M. Robinson and K. Schmidt (eds.), pp. 49–64.
6. C. A. Ellis and S. J. Gibbs, "Concurrency control in group systems," *SIGMOD RECORD* **18**, 1989, pp. 399–407.
7. U. Kelter, "Deadlock-freedom of large transactions in object management systems," *Information Systems* **14**, 1989, pp. 175–180.
8. U. Kelter, "Group Paradigms in Discretionary Access Controls for Object Management Systems," *Software Engineering Environments*, Fred Long (ed.), LNCS #467, Springer-Verlag, Berlin, 1989, pp. 219–233.
9. C. Hübel, W. Käfer and B. Sutter, "Controlling cooperation through design-object specification—A database-oriented approach," Universität Kaiserslautern, Forschungsbericht 21/92, Kaiserslautern, Germany, January 1992.
10. H. Balsters, R. A. de By and R. Zicari, "Typed sets as a basis for object-oriented database schemas," *Proc. Seventh European Conference on Object-Oriented Programming*, July 1993, Kaiserslautern, Germany, LNCS #707, Springer-Verlag, pp. 161–184.
11. H. Balsters and M. M. Fokkinga, "Subtyping can have a simple semantics," *Theoretical Computer Science* **87**, September 1991, pp. 81–96.
12. R. Bal, H. Balsters, R. A. de By, A. Bosschaart, J. Flokstra, M. van Keulen, J. Skowronek and B. Termorshuizen, "The TM Manual; version 2.0, revision e," Universiteit Twente, Technical report IMPRESS / UT-TECH-T79-001-R2, Enschede, The

Netherlands, June 1995.
13. T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems* **14**, 1987, pp. 25–59.
14. C. A. Vissers, G. Scollo, M. van Sinderen and E. Brinksma, "Specification styles in distributed systems design and verification," *Theoretical Computer Science* **89**, 1991, pp. 179–206.
15. ISO Standard, "Information processing systems—Open Systems—LOTOS—A formal description technique based on the temporal ordering of observational behaviour," International Standard ISO 8807:1988(E).
16. F. J. Faase, S. J. Even and R. A. de By, "Deliverable IV.3: An introduction to CoCoA," Universiteit Twente, Report TC/REP/UT/D4-3/033, ESPRIT LTR Project TRANSCOOP (8012), Enschede, The Netherlands, February 1996.
17. L. Cardelli, "A semantics of multiple inheritance," *Information and Computing* **76**, 1988, pp. 138–164.
18. H. J. Steenhagen, P. M. G. Apers, H. M. Blanken and R. A. de By, "From nested-loop to join queries in OODB," *Proc. 20th International Conference on Very Large Databases*, Santiago, Chile, September 1994, pp. 618–629.
19. D. Spelt, "A Proof Tool for TM," M.Sc. Thesis, Universiteit Twente, Enschede, The Netherlands, July 1995.
20. M. Caneve and E. Salvatori (eds.), "The LITE User Manual," The LOTOSPHERE Consortium, Lotosphere Deliverable Lo/WP2/N0034/V08, ESPRIT 2304, March 1992.
21. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1, EATCS Monographs on Theoretical Computer Science, 6*, Springer-Verlag, Berlin, 1985.
22. S. Even and F. Faase, "Deliverable IV.1: Merge options for LOTOS and TM," Universiteit Twente, Report TC/REP/UT/D4-1/009, ESPRIT LTR Project TRANSCOOP (8012), Enschede, The Netherlands, October 1994.
23. M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch and P. Muth, "Towards a cooperative transaction model: The cooperative activity model," *Proc. 21st International Conference on Very Large Databases*, Zurich, Switzerland, September 1995.
24. H. van Benthem, "Safety, Sets & TM," M. Sc. Thesis, Universiteit Twente, Enschede, The Netherlands, February 1994.
25. E. LOTOS (ed.), Revised Working Draft on Enhancements to LOTOS, May 1995, ISO/IEC JTC1/SC21, Working Group 7. Available via ftp: ftp.dit.upm.es, /pub/lotos/elotos.
26. R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*, MIT Press, 1989.
27. A. Jeffrey, G. Leduc and C. Pecheur (eds.), "A discussion of data types in E-LOTOS," document of ISO/IEC JTC1/SC21, Working Group 7, *Enhancements to LOTOS*, Paris meeting, February 1995.
28. A. Prakash and H. S. Shim, "DistView: Support for building efficient collaborative applications using replicated objects," *Proc. Fifth Conference on Computer-Supported Cooperative Work*, Chapel Hill, October 1994.
29. U. Asklund, "Identifying conflicts during structural merge," *Proc. NWPER'94, Nordic Workshop on Programming Environment Research*, Lund, Sweden, June 1994.
30. S. Minör and B. Magnusson, "A model for semi-(a)synchronous collaborative editing," *Proc. Third European Conference on Computer-Supported Cooperative Work—ECSCW'93*, Milano, Italy, 1993.
31. B. Magnusson, U. Asklund and S. Minör, "Fine-grained revision control for collaborative software development," *Proc. ACM SIGSOFT'93 Symposium on the Foundations of Software Engineering*, Los Angeles, December 1993.
32. T. Olsson, "Group awareness using fine-grained revision control," *Proc. NWPER'94, Nordic Workshop on Programming Environment Research*, Lund, Sweden, June 1994.

33. A. Prakash and M. J. Knister, "Undoing actions in collaborative work," *Proc. Fourth Conference on Computer-Supported Cooperative Work*, Toronto, October 1992, pp. 273–280.
34. J. Rekers and I. Sprinkhuizen-Kuyper, "A LOTOS specification of a CSCW tool," *Proc. Design of Computer supported cooperative work and groupware systems*, Schearding, Austria, June 1993.
35. C. G. Wolf, J. R. Rhyne and L. K. Briggs, "Communication and information retrieval with a pen-based meeting support tool," Technical Report RC 17842, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1992.
36. R. A. de By and H. J. Steenhagen, "Interfacing Heterogeneous Systems through Functionally specified Transactions," *Proc. IFIP DS–5 Conference on Semantics of Interoperable Database Systems*, Lorne, Victoria, Australia, November 1992.
37. R. A. de By, S. J. Even and P. A. C. Verkoulen, "Functionally Specified Distributed Transactions in Co-operative Scenarios," *Proc. Research Issues in Data Engineering—Distributed Object Management (RIDE-DOM)*, March 1995, Taipei, Taiwan (IEEE Computer Society Press).
38. J. Vis, E. Brinksma and R. A. de By, "Formal specification of distributed information systems, extended position paper," *Proc. CAiSE'94-workshop on Formal Descriptions of Distributed Information Systems*, Utrecht, The Netherlands, 1994.
39. J. Vis, E. Brinksma and R. A. de By, "Semantic integration of LOTOS process and TM database specification," *Proc. COST 247, Verification and Validation Methods for Formal Descriptions*, Humboldt Universität, Informatik-Bericht Nr. 45, Berlin, Germany, February 1995.
40. L. Cardelli, "Oblique: A language with distributed scope," Report n.122, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, November 1994.
41. J. Flokstra, M. van Keulen and J. Skowronek, "The IMPRESS DDT: A database design toolbox based on a formal specification language," *Proc. of ACM-SIGMOD 1994 International Conference on Management of Data*, Minneapolis, MN, May 1994.
42. H. Eertink, *Simulation Techniques for the Validation of LOTOS Specifications*, Ph. D. Dissertation, Universiteit Twente, Enschede, The Netherlands, 1994.
43. H. Eertink, "SMILE User Manual, release 4.0," Universiteit Twente, Enschede, The Netherlands, December 1993.
44. S. J. Even, F. J. Faase, R. A. de By and O. Pihlajamaa, "Deliverable IV.4: The TRANSCOOP Specification Environment," Universiteit Twente, Report TC/REP/UT/D4-4/032, ESPRIT LTR Project TRANSCOOP (8012), Enschede, The Netherlands, May 1996.