Functionally Specified Distributed Transactions in Co-operative Scenarios*

Rolf A. de By Susan J. Even Peter A. C. Verkoulen†

Centre of Telematics and Information Technology–University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract

We address the problem of specifying co-operative, distributed transactions in a manner that can be subject to verification and testing. Our approach combines the process-algebraic language LOTOS and the object-oriented database modelling language TM to obtain a clear and formal protocol for distributed database transactions meant to describe co-operation scenarios. We argue that a separation of concerns, namely interaction of database applications on the one hand and data modelling on the other hand, results in a practical, modular approach that is formally well-founded. An advantage of this is that we may vary over transaction models to support the language combination.

1 Introduction

In application areas like computer-supported co-operative work (CSCW) there is a strong need for primitives to describe high-level co-operation scenarios that can be implemented in a verifiable way. These scenarios share a set of characteristics that make them intrinsically complex to describe and implement. Typically, the area is identified by a high level of interaction between agents, combining complex data communication with both local and shared data resources. It is our firm belief that such scenarios can only be described by paying due attention both to process behaviour and data modelling.

In the TransCoop project (ESPRIT BRA 8012) we are investigating the combination of LOTOS as a process modelling language with TM as an object-oriented data modelling language. Within TransCoop, we also study how specifications obtained in this language combination can be mapped onto database platforms that feature advanced transaction models, like the advanced transaction models of [1–6]. We are studying co-operative work applications found in industrial environments with the aim of identifying specification language requirements for this field. Preliminary work was described in [7]. The goals that we have in mind in performing this work are the following:

- **orthogonality** of the LOTOS/TM language combination. The integration will be one of co-existence, with the two languages used to describe different aspects of a co-operative scenario.
- **orthogonality** of database functionality and inter-database communication protocols as a result of the above. This requires explicit synchronization with the DBMS to obtain query results and perform database updates.
- **genericity** of the system structure description such that the database functionality remains modular, and one may vary over used transaction models.
- **abstraction** away from implementational details with which applications should not deal.
- **parameterizability** over architectural features that would identify classes of co-operation scenarios.

In this short paper, we present an overview of our initial results towards a specification framework which addresses the above goals. Section 2 discusses how we go about combining LOTOS and TM, and discusses their use in a three step method. We then briefly discuss the two languages in more detail in Section 3. Section 4 describes an essential primitive for co-operation scenario definition, called the LTM event. The full paper [8] gives example specifications and provides a formal semantics of LTM events. Section 5 summarizes the novelties of our approach and identifies further issues that we want to address in this area.

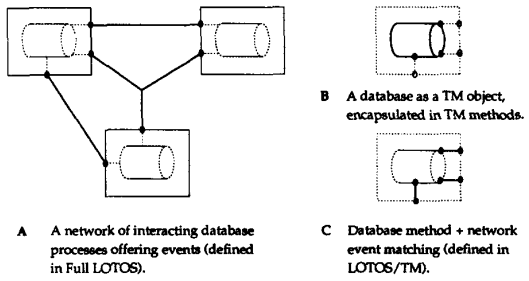
2 The specification framework

The essence and intention of combining LOTOS and TM is to arrive at a pleasing 'new' specification language for distributed information systems that is both process-based and state-based. We view the definition of a network of distributed information systems as an interconnected network

*This work is carried out in the ESPRIT project TRANSCOOP (EP8012) which is partially funded by the Commission of the European Communities. The partners in the TRANSCOOP project are GMD (Germany), University of Twente (The Netherlands), and VTT (Finland).

†Our respective email addresses are [deby, seven, verkoule]@cs.utwente.nl

0-8186-7056-8/95 $04.00 © 1995 IEEE



A A network of interacting database processes offering events (defined in Full LOTOS).

B A database as a TM object, encapsulated in TM methods.

C Database method + network event matching (defined in LOTOS/TM).

Figure 1: The specification framework.

of LOTOS processes which each have their local database. There are three steps in the complete specification of such a network, as illustrated in Figure 1. The behaviour of the network itself can be completely described in Full LOTOS (see Figure 1A). The bullets indicate event gates that the information systems offer; the lines indicate which communication is possible.

A second step in the complete description is to define the requested functionality for each database system. This is illustrated in Figure 1B. We use the data model TM to provide method definitions that will in the next step be matched against event gates.

The interesting part of the language combination is where the interface between the two formalisms lies. In terms of completing the system definition, we have to identify how network events are matched to local database methods, thus essentially defining the interface between the database and the communication network. This is the really new part of the language combination, and it is illustrated in Figure 1C. The specification is discussed in Section 4.

3 Language descriptions

In this section, we summarize the important features of TM and LOTOS.

3.1 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 [9,10]. It is formally founded in a typed lambda calculus allowing for subtyping and multiple inheritance, based on the ideas of [11]. We use it here for the description of the communicated data structures of the network, *and* for the description of the local database schemata.

Characteristic features of TM are the distinction between types and classes; support for object identity and complex objects; a three level method and constraint specification facility (object, class and database); and multiple inheritance of data structure, methods and constraints. A TM database is defined 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 ($\mathbb{L}\ldots$). Besides basic types, class names may be used in type specifications. Specialization between classes is denoted by the ISA-clause, which may identify several superclasses. A class inherits the attributes, constraints, and methods of its superclasses. In class specifications, constraints and methods may be specified at two different levels: the *object level* and the *class level.*

A TM database specification is organized in terms of class and sort definitions. Each class (sort) definition identifies the type of its instances, rules that the instances have to obey, and methods to which they respond. 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. 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.

Together with a collection of class and sort 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. The variable will then represent the class's so-called extension.) 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 rules 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. For our discussion, it is important to know that methods come in two flavours: *retrieval methods* and *update methods*. Retrieval methods (or queries) take the database and retrieve some information from it. Update methods (or transactions) take the database and change its state. Obviously, both forms of methods may require parameters.

A tool-set for TM, including graphical interface, type checker, prototype generator and compiler is under development in the ESPRIT project IMPRESS (6355).

3.2 Basic LOTOS

LOTOS is a specification language that was designed for the formal description of distributed, concurrent systems [12–14]. LOTOS is an ISO standard (ISO 8807), which makes it a more than viable candidate for the use we give it here. 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 Section 3.3.)

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 an interaction is called an event. (Sub)processes are defined in terms of behaviour expressions; behaviour expressions bottom out in event names (or gates).

Possible forms of Basic LOTOS behaviour expressions B are given in Table 2, which we have adopted from [12]. The **stop** process is the process incapable of any (inter)action. Action prefix allows to express that an action occurs before some behaviour. The general case parallel composition identifies n gates on which the behaviours $B1$ and $B2$ have to synchronize. Pure interleaving is a special case, namely where $n = 0$. Likewise, full synchronization is the case where the set of gates $\{ g_1, \ldots, g_n \}$ is the union of the sets of gates offered by $B1$ and $B2$. Hiding allows to turn actions at the mentioned gates into internal actions of the process. Behaviour 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 behaviour expressions. Disabling, finally, allows to express that a normal behaviour $B1$ may, at any moment, be disrupted by another behaviour $B2$.

A good introduction to the operational semantics of LOTOS is found in [12]. A LOTOS tool-set environment for simulation, compilation and prototype generation is described in [15].

3.3 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; here, we will instead use TM as the data language. This section looks at constructs of Full LOTOS which involve value expressions and therefore will be at the core of the LOTOS/TM language combination.

In Full LOTOS, an event name may be followed by value and variable declarations, together also called *event attributes*. An event name together with its attributes is usually referred to as a *structured event*. A value declaration looks like $!e$, where e is an allowed value expression in the data language. A variable declaration takes the form $?x : \sigma$, where x is a variable identifier and σ is a data type. An example LOTOS structured event schema is:

$$\texttt{gate}\ !e_1 \ \ldots\ !e_n\ ?x_1{:}\sigma_1\ \ldots\ ?x_m{:}\sigma_m\ [\mathit{cond}]$$

where *cond* is an expression of type `bool`, e_1 up to e_n are arbitrary TM value expressions (queries, as we shall see in Section 4), and $x_1{:}\sigma_1$ up to $x_m{:}\sigma_m$ are typed variable declarations. A condition may be used to constrain possible values for data synchronization.

There are other possible occurrences of TM expressions in Full LOTOS which are not part of structured events. These are: actual parameters to process instantiations other than actual gate names, guards as conditions on possible behaviour, and the **let**-construct. Each of these, we will briefly describe.

A process instantiation takes the form

$$\mathrm{p}[g_1, \ldots, g_n]\ (e_1, \ldots, e_m)$$

where the g_i are actual gate names and the e_i stand for actual values provided to the process. A guard is a condition (i.e., a boolean TM expression) that needs to be fulfilled to make a behaviour possible; if the guard is not fulfilled the following guarded behaviour expression is equivalent to the **stop** process:

$$[\texttt{guard}] \rightarrow B$$

Finally, the **let** construct introduces (typed) abbreviations for value expressions in a behaviour expression:

$$\textbf{let}\ x_1{:}\sigma_1 = e_1,\ \ldots,\ x_n{:}\sigma_n = e_n\ \textbf{in}\ B$$

In each of the above cases, TM method invocations may be part of the expression. Such invocations will be taken to be *retrieval methods* against the process's local TM database. LOTOS imposes no evaluation order on e_1 to e_n, and favourably, TM retrieval methods have no side-effects on the database.

4 Co-operative transaction events

Section 3.3 highlighted the value-based constructs of Full LOTOS where integration with TM will arise—i.e., those places where the two languages will meet. The most interesting of these constructs is the structured event, as this is the point where the idea of co-operative transactions becomes apparent. Intuitively, a LOTOS/TM event describes a complex communication event in which local database updates are dependent on the results of queries against remote databases. The event might be triggered by an application process and will usually depend upon synchronization with other agents. In this section, we outline the semantics of a LOTOS/TM event.

4.1 LOTOS/TM events for co-operative updates

A LOTOS/TM event (LTM event for short) describes the behaviour that occurs at the interface between a TM

expression name	syntax B ::=	expression name	syntax B ::=
inaction	**stop**	parallel composition	
action prefix		– general case	$B1 \,\|[g_1, \ldots, g_n]\|\, B2$
– unobservable (internal)	$\mathbf{i}; B$	– pure interleaving	$B1 \,\|\|\|\, B2$
– observable	$g; B$	– full synchronization	$B1 \,\|\|\, B2$
choice	$B1 [] B2$	hiding	**hide** $g_1, \ldots, g_n$ **in** B
process instantiation	$p[g_1, \ldots, g_n]$	enabling	$B1 >> B2$
successful termination	**exit**	disabling	$B1 [> B2$

Figure 2: Basic LOTOS behaviour expression syntax.

database and its encapsulating LOTOS process. We want to rigourously define what event occurrence at the interface means. An LTM event either happens or does not happen—in other words it is atomic. If it does not happen, no pure LOTOS event has occurred and no TM updates have been issued. Locally, we view an LTM event as a triple (U_{pre}, E, U_{post}) where E is a LOTOS event. The motivation for this view stems from the observation that an event occurrence typically coincides with a state change of the process. But to cover situations of triggering and non-triggering behaviour, we identify two optional state changes: one immediately before, and one immediately after the actual event occurrence.

Intuitively, U_{pre} will be used for setting up any local workspace involved in the event; U_{post} happens *after* the LOTOS structured event. Methods U_{pre} and U_{post} are called the event's pre- and post-update. Both are in fact optional, and this may be indicated by the identity update. (If there is no database associated with the process in which the event expression occurs, then both U_{pre} and U_{post} will be the identity.) Although TM has no formal transaction notion—updates are just state transitions of the database—we define it to be as follows. An update *commits* if and only if all database constraints are valid for the post-state of the database; otherwise it *aborts*. We assume the availability of a method *dbcons* that functions as a test on database constraint integrity [9]. After commit of a TM update, the database state has usually changed.

Figure 3 captures the intuitive meaning of an LTM event.

The transactional aspects of an LTM event are summarized as follows:

1. Any TM method invocation occurring in the condition or in a value expression is interpreted as a *retrieval method.*

2. The U_{pre} update method associated to the event is invoked. If it aborts, the complete LOTOS/TM event does not occur. Otherwise,

3. A synchronization attempt on the LOTOS event may take place, evaluating any value expressions against the newly obtained database state, i.e. the post-state of U_{pre}. If the attempt does not succeed, the U_{pre} update is rolled back, and the overall LOTOS/TM event does not occur. Otherwise,

4. The U_{post} update method associated to the event is invoked. If it commits, we obtain another new database state and the overall event succeeds. If U_{post} aborts, U_{pre} is rolled back and the overall event does not occur.

From the more abstract point of view of application specification, a LOTOS/TM event is an atomic primitive. Several forms of such events can be identified, and their implementation is discussed in the full paper [8].

4.2 LOTOS/TM event declaration

The relationship between LOTOS/TM events and possible pre- and post- TM database update methods will be provided by user-specified *event declarations* with the following form:

LTM_event `process·gate` **is**
`gate` $!e_1\ ?x_1{:}\tau_1 \ldots !e_n\ ?x_n{:}\tau_n$
[triggered by U_{pre}**]**
[with update $U_{post}(c_1, \ldots, c_\ell)$**]**

The declaration includes the LOTOS event specification, together with expressions for its optional pre- and post- TM method invocations. An LTM event environment binding LTM event names to LTM event declarations will be used in the evaluation of the top-level LOTOS network specification.

The naming convention for an LTM event will be to juxtapose its gate name with the process name in which the event is found (i.e., its context). Although a process name is unique for a specification, it does not uniquely determine a context for the gate name: a gate name may appear multiple times within the process body, and each use may be syntactically different in terms of the number, type and form (whether ! or ?) of its attributes. We will distinguish such

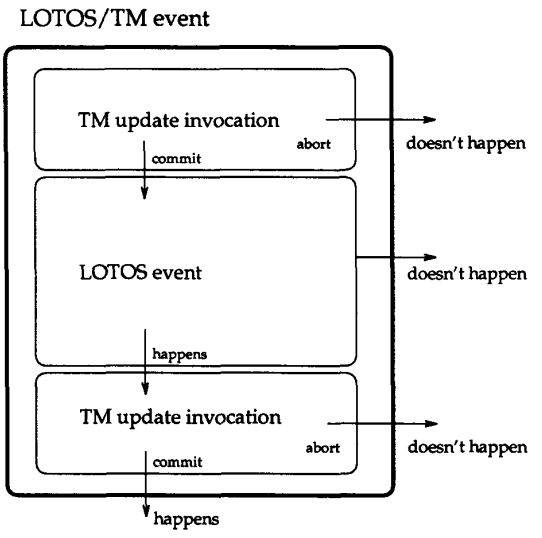


Figure 3: Transactional aspects of an LTM event.

multiple occurrences of the same gate within a process by a numerical index when needed.

The signature associated with U_{post} in the TM database schema may differ significantly from the sequence of TM types on the gate listed above. We must therefore provide a mapping from the gate attributes to the actual parameters of the TM method invocation. This is done schematically by the above syntax. Expressions c_1 up to c_ℓ can include any of the expressions and variable declarations that appear as gate attributes, possibly as subexpressions. The attributes to be used as method parameters are simply written as they appear in the event specification, for example, as in the expression $U_{post}(e_3, x_5, e_2)$. We require that c_1 through c_ℓ contain no free identifiers other than x_1 through x_n. Results of queries on other, non-local databases involved in the LOTOS/TM event may thus serve as input to U_{post}.

4.3 LOTOS/TM event evaluation

Consider the following structured event schema:

$$\texttt{gate}\ !e_1\ \ldots !e_n\ ?x_1{:}\sigma_1\ \ldots ?x_m{:}\sigma_m$$

The rôles of the different kinds of gate attributes in the computation can be understood as follows:

- a value declaration e_i
  1. may invoke retrieval methods,
  2. may impose constraints on which processes can synchronize with the current one, and
  3. may be used as an actual parameter in the optional post-update method invocation.
- a variable declaration x_i
  1. is bound to a value communicated as a result of the synchronization, and
  2. this value may be used as actual parameter in the optional post-update method invocation.

Evaluation of an LTM event imposes the following dependencies on value communication and synchronization. U_{pre} happens *before* the LOTOS structured event; it does not require as input any values communicated during the event. U_{pre} might, however, update the database such that it affects the retrieval methods invoked during the evaluation of TM expressions that appear as value declarations on `gate`. After synchronization takes place, prior to the evaluation of U_{post}, any variable declarations on `gate` will have been bound to values. These values can in turn serve as input arguments to method U_{post}. All required input values to the U_{post} update method must have been obtained prior to its evaluation, either as a result of the synchronization event `gate`, or as a result of TM queries on the local database.

It follows from our approach that synchronization becomes dependent on the state of the process's local database, simply because retrieval method invocations may be used. Additionally, values communicated as a result of synchronization affect subsequent database updates, and thus these updates become synchronization-dependent on the states of other databases. But there is a catch here: the event(s) with which our original event synchronizes may or may not belong to the same process (and hence database). We shall assume process and database descriptions to be decomposed in such a way that a process does not require synchronization with itself.

5 Conclusions and Further work

In the paper, we have illustrated how a combination of the LOTOS and TM languages can be used to describe distributed object services. It turns out that a number of advantageous characteristics are inherent in our approach:

1. the languages can be combined in a purely orthogonal manner, requiring no syntax changes to either, but only some minute additional interface definitions that can be kept separate from the specifications;
2. specifications in the language combination nicely distinguish between database network behaviour and monolithic database functionality;
3. the languages accommodate generic system structure definitions, and this allows us to localize DBMS functionality and let it vary with specific choices for advanced transaction models; this set-up is just one—albeit important—example of an architectural choice made parameterisable;

4. the language combination allows abstract specifications of systems as well as primitives (macros) for high-level modelling of co-operation scenarios; the paper identifies a first elementary primitive, the LTM event, and the full paper gives its semantics in LOTOS.

Our continuing work will investigate other co-operation scenario requirements which we did not address here; such as time and space keeping agents, and database support for partial aborts or commits. The latter requirement will be studied in the context of an open nested transaction model. In line with this, we will attempt to provide more concurrency in the database functionality, for instance by decomposing the *dbcons* function into functions that take into account the nature of the updates (cf. semantic concurrency control).

We have demonstrated the possibility of meeting some of the goals that we identified in the introduction with our proposed specification framework. We hope to discover modular primitives along the lines of the LTM event, which can be composed to achieve the same results in a more general way. Our future work will unfold in the direction of a complete language definition for the combined language, tool support for the language, and implementation mapping to advanced database transaction models.

6 Acknowledgements

We thank Pieter Oude Egberink for carrying out a preliminary case study, and thank Frans Faase and Jan Vis for discussing with us the ideas presented in this paper.

References

[1] Gerhard Weikum, Andrew Deacon, Werner Schaad & Hans Schek, "Open nested transactions in federated database systems," *IEEE Bulletin on Data Engineering* 1 (June 1993), 4–7.

[2] Omran Bukhres, Ahmed Elmagarmid & Eva Kuhn, "Implementation of the Flex transaction model," *IEEE Bulletin on Data Engineering* 1 (June 1993), 28–31.

[3] Peter Muth, Thomas C. Rakow, Wolfgang Klas & Erich J. Neuhold, "A transaction model for an open publication environment," in *Database Transaction Models for Advanced Applications*, Ahmed K. Elmagarmid, ed., Morgan Kaufmann Publishers, San Mateo, CA, 1992, 159–218.

[4] Jari Veijalainen, Frank Eliassen & Bernhard Holtkamp, "The S-transaction model," in *Database Transaction Models for Advanced Applications*, Ahmed K. Elmagarmid, ed., Morgan Kaufmann Publishers, San Mateo, CA, 1992, 468–513.

[5] Helmut Wächter & Andreas Reuter, "The ConTract model," in *Database Transaction Models for Advanced Applications*, Ahmed K. Elmagarmid, ed., Morgan Kaufmann Publishers, San Mateo, CA, 1992, 219–264.

[6] Panos K. Chrysanthis & Krithi Ramamritham, "ACTA: The SAGA continues," in *Database Transaction Models for Advanced Applications*, Ahmed K. Elmagarmid, ed., Morgan Kaufmann Publishers, San Mateo, CA, 1992, 349–398.

[7] Rolf A. de By & Hennie J. Steenhagen, "Interfacing Heterogeneous Systems through Functionally specified Transactions," in *IFIP DS–5 Conference on Semantics of Interoperable Database Systems, Lorne, Victoria, Australia, November 16–20, 1992 #2*, David K. Hsiao, Erich J. Neuhold & Ron Sacks-Davis, eds., IFIP & CITRI, 1992, 38–45.

[8] R. A. de By, Susan J. Even & Peter A. C. Verkoulen, "Functionally specified distributed transactions in co-operative scenarios," University of Twente, Memorandum INF–94–72, Enschede, 1994, 18 pp.

[9] H. Balsters, R. A. de By & R. Zicari, "Typed sets as a basis for object-oriented database schemas," in *Proceedings Seventh European Conference on Object-Oriented Programming, July 26–30, 1993, Kaiserslautern, Germany, LNCS #707*, Oscar M. Nierstrasz, ed., Springer-Verlag, New York–Heidelberg–Berlin, 1993, 161–184.

[10] René Bal, Herman Balsters, Rolf A. de By, Alexander Bosschaart, Jan Flokstra, Maurice van Keulen, Jacek Skowronek & Bart Termorshuizen, "The TM Manual," Universiteit Twente, Technical report IMPRESS/UT-TECH-T79-001-R2, Enschede, The Netherlands, 1993.

[11] Luca Cardelli, "A semantics of multiple inheritance," *Information and Computation* 76 (1988), 138–164.

[12] Tommaso Bolognesi & Ed Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems* 14 (1987), 25–59.

[13] Chris A. Vissers, Giuseppe Scollo, Marten van Sinderen & Ed Brinksma, "Specification styles in distributed systems design and verification," *Theoretical Computer Science* 89 (1991), 179–206.

[14] ISO–Information Processing Systems–Open Systems Interconnection, *LOTOS–A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, DIS 8807, 1987.

[15] Maurizio Caneve & Elena Salvatori, eds., "Lite User Manual," LOTOSPHERE consortium, Lotosphere deliverable Lo/WP2/N0034/V08, ESPRIT 2304, March 30, 1992.