INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# Requirement Capture, Formal Description and Verification of an Invoicing System

Mihaela Sighireanu, Kenneth J. Turner

N˚ 3575

THÈME 1

Rapport de recherche

ISSN 0249-6399

# Requirement Capture, Formal Description and Verification of an Invoicing System

Mihaela Sighireanu*, Kenneth J. Turner†

Thème 1 — Réseaux et systèmes
Projet VASY Recherche et Applications

Rapport de recherche n3575 — — 32 pages

**Abstract:** The Invoicing case study is a typical business system proposed by Henri Habrias as a common example for a contest on the capacity of particular formal methods to capture requirements from the client. For this, the case study is informally described by half a page of English text. In this report, we use the formal description technique LOTOS for requirement capture, formal description and verification of the Invoicing case study. First, we analyse and interpret the informal requirements of the case study using the LOTOS approach for description of systems. This leads to a set of twenty questions about the informal description. By answering to these questions, we obtain a high-level specification architecture that can be formalised. Then, we present the formal description of the case study in LOTOS and, for comparison, in E-LOTOS, the new version of LOTOS currently being standardized. Since LOTOS allows a balance to be struck between process-oriented and data-oriented modeling, descriptions in both styles are given. After that, we verify the LOTOS descriptions by model-checking using the CADP (CÆSAR/ALDÉBARAN) toolbox. The underlying Labelled Transition System (LTS) models corresponding to various scenarios are generated using the CÆSAR compiler. We push further the analysis of the case study by formalizing in temporal logic six properties of the system. We verify these properties on the LTS models using the XTL model-checker. Finally, we study the equivalence of the process-oriented and data-oriented descriptions using the ALDÉBARAN tool.

**Key-words:** E-LOTOS (Enhancements to LOTOS), Formal Methods, Formal Description Techniques, Invoicing System, Labelled Transition Systems (LTS), LOTOS, (Bi)Simulation relation, Temporal Logic, Verification.

*(Résumé : tsvp)*

A previous version of this report is available as two separate papers: "The Invoicing Case Study in (E-)LOTOS" by Kenneth J. Turner, and "Model-checking Validation of the LOTOS Descriptions of the Invoicing Case Study" by Mihaela Sighireanu, in Henri Habrias, editor, *Proceedings of the First International Workshop on Comparing System Specification Techniques, CSST'98, Nantes, March 26–27 1998.*

* VASY, INRIA Rhône-Alpes, France, e-mail: `Mihaela.Sighireanu@inria.fr`
† Department of Computing Science and Mathematics, University of Stirling, Scotland, e-mail: `kjt@cs.stir.ac.uk`

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

# Ingénierie des besoins, description formelle et vérification d'un système de facturation

**Résumé :** L'étude de cas "Facturation" est un système de gestion classique proposé par Henri Habrias comme un exemple commun pour un débat comparatif sur la capacité des différentes méthodes formelles à analyser les besoins des clients. Pour cette raison, le système est décrit d'une façon informelle, dans une demi-page de texte en anglais. Dans ce rapport, nous utilisons la technique de description formelle LOTOS pour l'ingénierie des besoins, la description formelle et la vérification de cette étude de cas. Premièrement, nous analysons et nous interprétons la description informelle des besoins en utilisant l'approche proposée par LOTOS pour la description des systèmes. Ceci nous conduit à poser vingt questions sur la description informelle. En répondant à ces questions, nous obtenons une spécification de haut niveau qui peut être formalisée. Ensuite, nous présentons la description formelle du système en LOTOS et, pour comparaison, en E-LOTOS, la nouvelle version de LOTOS en cours de normalisation. Puisque LOTOS offre la possibilité d'utiliser plusieurs styles de description, deux descriptions sont présentées : une description orientée processus et une description orientée données. Nous vérifions les descriptions LOTOS en utilisant l'approche basée sur les modèles (*model-checking*) au moyen de la boîte à outils CADP (CÆSAR/ALDÉBARAN). Les modèles système de transitions étiquetées (STE) correspondant aux différents scénarios sont générés à l'aide du compilateur CÆSAR. Nous avançons l'analyse de l'étude de cas en formalisant en logique temporelle six propriétés du système. Nous vérifions ces propriétés sur les modèles STE au moyen de l'évaluateur XTL. Finalement, nous étudions l'équivalence entre les descriptions orientées processus et les descriptions orientées données au moyen de l'outil ALDÉBARAN.

**Mots-clé :** E-LOTOS (Enhancements to LOTOS), Facturation, Logique temporelle, LOTOS, Méthodes formelles, Relation de (bi)simulation, Systèmes de transitions étiquetées (STE), Techniques de description formelle, Vérification.

# 1 Introduction

A large number of formal specification and verification methods have been developed in the last twenty years. The Formal Methods site of the Oxford University[1] contains seventy-five references to such methods. Several case studies have been proposed in order to test the suitability of a particular method for formal specification and verification of an engineering discipline or to compare different methods from a specific point of view. Henri Habrias proposed a case such study, namely the Invoicing system.

The Invoicing system is a typical business system. We include below its informal description, as proposed by Henri Habrias [AAH98]:

> *The subject is to invoice orders. To invoice is to change the state of an order (to change it from the state "pending" to "invoiced").*
>
> *On an order, we have one and one only reference to an ordered product of a certain quantity. The quantity can be different from other orders. The same reference can be ordered on several different orders. The state of the order will be changed into "invoiced" if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product.*
>
> *You have to consider the two following cases:*
>
> **Case 1:** *All the ordered references are references in stock. The stock or the set of orders may vary due to the entry of new orders or cancelled orders, or due to having a new entry of quantities of products in stock at the warehouse.*
>
> *But, we do not have to take these entries into account. This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.*
>
> **Case 2:** *You do have to take into account the entries of:*
>
> - *new orders;*
> - *cancellations of orders;*
> - *entries of quantities in the stock.*

The declared goal of the Invoicing case study is to evaluate the capacity of particular formal methods to capture requirements from the client. This evaluation is obtained by asking the specifier using a particular formal method for formal specification and verification of the case study to consider the following concerns:

- What are the questions raised by the formalization of the informal description?
- What are the solutions suggested or imposed by the method to these questions?
- How these solutions may be verified to be consistent?

In this report, we address the Invoicing case study using the LOTOS [ISO88] and E-LOTOS [Que98] formal description techniques and the CADP [GJM+97] protocol engineering toolbox.

The report is organized as follows. Section 2 introduces briefly the LOTOS and E-LOTOS languages and the LOTOS approach for the requirement capture. Section 3 presents the questions arisen when trying to formalize the Invoicing case study, and the revised problem statement obtained by answering to these questions. Section 4 presents the LOTOS and E-LOTOS formal descriptions of the Invoicing case study using the process-oriented and the data-oriented specification styles. Section 5 presents the testing and verification issues and introduces the CADP protocol engineering toolbox. Section 6 presents the generation of the LTS models corresponding to the LOTOS descriptions. Sections 7 and 8 describe the verifications performed on these models by means of temporal logic and bisimulation equivalences. Section 9 gives some concluding remarks.

[1] `http://www.comlab.ox.ac.uk/archive/formal-methods.html`

# 2 The LOTOS Approach

## 2.1 The ISO Language LOTOS

LOTOS [ISO88] is a standardized Formal Description Technique intended for the specification of communication protocols and distributed systems. Several tutorials for LOTOS are available, e.g. [BB88, Tur93b].

The design of LOTOS was motivated by the need for a language with a high abstraction level and a strong mathematical basis, which could be used for the description and analysis of complex systems. As a design choice, LOTOS consists of two "orthogonal" sub-languages:

**The data part** of LOTOS is dedicated to the description of data structures. It is based on the well-known theory of algebraic abstract data types [Gut77], more specifically on the ACTONE specification language [dMRV92].

**The control part** of LOTOS is based on the process algebra approach for concurrency, and appears to combine the best features of CCS[2] [Mil89] and CSP[3] [Hoa85].

LOTOS has been applied to describe complex systems formally, for example: OSI TP[4] [ISO92, Annex H], MAA[5] [Mun91], FTAM[6] basic file protocol [LL95], etc. It has been mostly used to describe software systems, although there are recent attempts to use it for asynchronous hardware description [CGM+96].

A number of tools have been developed for LOTOS, covering user needs in the areas of simulation, compilation, test generation, and formal verification.

Despite these positive features, a revision of the LOTOS standard has been undertaken within ISO since 1993, because feedback from users indicated that the usefulness of LOTOS is limited by certain characteristics related both to technical capabilities and user-friendliness of the language. The last version of the ISO Committee Draft [Que98], appeared in April 1998, proposes a revised version of LOTOS, named E-LOTOS. Compared to LOTOS, the language defined in [Que98] introduces new features, from which we mention only those used in this report: modularity, functional (constructive) types, classical programming constructs, a controlled imperative style, gate typing. Since E-LOTOS standardization is ongoing, some of its constructs have still to be stabilised. Therefore, this paper is based on a variant of E-LOTOS, named LOTOS NT [GS98] defined at INRIA Rhône Alpes. In particular, the authors have assumed that a fully imperative semantics will be introduced. Among other things, this simplifies the specification of loops.

## 2.2 Requirement Capture in LOTOS

The LOTOS Position$^{TM}$ is to treat a system as a black box, and therefore to concentrate on its boundary, inputs and outputs. A LOTOS specifier will try to write a high-level specification of requirements, avoiding implementation-oriented concerns. The emphasis will be on specifying the partial ordering of (observable) events. Other factors that influence the approach include the balance chosen between processes and data in the specification, and the choice of specification style (if one is explicitly adopted). Various methods have been investigated for LOTOS, e.g. [BLV95, Tur90, GR90], but because the case study was so small, the authors followed only general LOTOS principles:

- delimit the boundary of the system to be specified;
- define the interfaces of the system (inputs, outputs, parameters);
- define the functionality of the system (the relationship among inputs and outputs);
- for incomplete requirements, choose an abstract or simple interpretation that will give some freedom later for adopting a more specific interpretation.

[2] Calculus of Communicating Systems
[3] Communicating Sequential Processes
[4] Distributed Transaction Processing
[5] Message Authentication Algorithm
[6] File Transfer, Access and Management

LOTOS is a constructive specification language: any specification will exhibit some structure (usually hierarchic, though a monolithic style is also possible). The subject of specification style has been investigated in considerable depth for LOTOS [Cla92, FLS97, Que90, SL93, Tur93a, Tur97a, Tur97b, TS95, Eij89, Sin89, VSS91, VSSB90]. Indeed it might be fairly said that LOTOS specifiers are pre-occupied with specification style! The choice of an appropriate style for specifying requirements has a big impact on how the specification is structured. Another way of putting this is to say that LOTOS specifiers care about the high-level architecture[7] of a system. Several LOTOS workers have considered general "quality" principles for specification architecture [Sco93, Tur97c].

Because LOTOS combines a data type language with a process algebra, the specifier must choose an appropriate balance when using these two aspects of the language [Led87]. This partly depends on the preferred specification style, partly on the intended use of the specification (e.g., for analysis or refinement), and partly on the application. Some applications focus on the representation and manipulation of data (e.g., a database), and so are more naturally specified using the data part of LOTOS. Other applications focus on dynamic (reactive) behaviour, and so are more naturally specified using the process algebra part of LOTOS.

A LOTOS-based approach to requirement capture raises the following kinds of questions:

**Environment:** Who are the users of the system? What is the context of the system? What is the boundary of the system? What functions can the system rely on in the environment?

**Interfaces:** What are the interfaces to the environment? What are the data flows into and out of the system? What is the structure and content of these data flows?

**Functionality:** What functions must the system perform? What is the relationship among inputs and outputs?

**Limitations:** What limits apply to system inputs, outputs and functions?

**Non-functional aspects:** What timing and performance aspects must be specified? What other organisational issues should be considered?

**Methodology:** How should the formal model be developed? Which specification style is appropriate? How should the specification be validated (by testing and/or verification)?

# 3 Analysis of Requirements

The act of formalisation typically raises many questions that would normally be discussed with the client. In a realistic situation, the systems analyst raises such questions with the client. This allows ambiguities, errors, and omissions in the requirements to be resolved. As in this case study, it is sometimes not possible to approach a client with questions. For example, it may be necessary to carry out a *post hoc* formalisation of something that already exists (e.g., a legacy system or an international standard). It was necessary for the authors to raise questions about the invoicing requirements and to answer them themselves in a sensible fashion.

In order to analyse and to interpret the informal description of the Invoicing case study presented in the introduction section, we assign to each informal requirement a reference (R$n$) which will be used in the remainder of this section. Using this assignment, the informal description becomes:

R0. General:

- R0.1 The subject is to invoice orders.
- R0.2 To invoice is to change the state of an order (to change it from the state "pending" to "invoiced").
- R0.3 On an order, we have one and only one reference to an ordered product of a certain quantity. The quantity can be different from other orders.
- R0.4 The same reference can be ordered on several different orders.
- R0.5 The state of the order will be changed to "invoiced" if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product.

[7] In the sense of [Tur97c], the architecture of a specification means its structure and style.

R1. Case 1:

R1.1 All the ordered references are in stock.

R1.2 The stock or the set of orders may vary due to the entry of new orders or cancelled orders, or due to having a new entry of quantities of products in stock at the warehouse. But we do not have to take these entries into account.

R1.3 This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.

R2. Case 2:

R2.1 You do have to take into account the entry of new orders, cancellations of orders, and entries of quantities in the stock.

## 3.1 Questions on Requirements

The following questions arise when trying to formalise the requirements for the Invoicing case study. The questions (Q$n$) are presented according to the classification in section 2.2, though they did not arise in this strict hierarchical order. As will be seen, the volume of questions is much greater than the informal problem statement!

**Environment**

Q1. How many users has the Invoicing system? This is not stated in the informal requirements. If there were only one user, it would not be necessary to identify orders (assuming that they were processed in sequence). If there were several users and it were necessary to issue invoices for users, it would be necessary to identify users or orders. Invoices would then have to carry an identification. Since the informal requirements do not ask the system to do anything (e.g., to produce an invoice or to deliver a product), this question does not arise.

**Interfaces**

Q2. For Case 1, "the stock and the set of orders are [...] given" (R1.3) means that the system does not directly accept stock or orders changes from the warehouse or user. It also means that the system has direct access to the current stock and orders. It follows that these must be maintained by some other sub-system.

Q3. At what point is the status of orders updated: when the stock or set of orders changes, or following a periodic check? If the former, how does the system know that there has been a change? If the latter, how frequently should the system check? The former interpretation is simpler and is therefore preferred. It follows that the system must be told of new stock or orders. This information thus becomes input to the system. The system must update the stock and orders, which by implication are stored within the system, since no outputs are mentioned. The system to be specified is thus an embedded sub-system of some larger system.

Q4. There is no indication of whether new stock or orders are notified individually or in batches to the system. For simplicity it is assumed that inputs occur individually.

Q5. There is the question of how invoicing is triggered, how the information is obtained, and how the status of orders is updated. Since the requirements imply that some internal agency manages the stock and orders, it is presumed that this agency supplies information to the system as required and triggers an update.

Q6. How is it possible to identify an order to be cancelled? The only sensible solution is if an order carries a reference that can subsequently be quoted in a cancellation. Other information such as the original product code or requested quantity (R0.3) would be redundant on cancellation and so are omitted.

Q7. Who, then, is responsible for creating an order reference? It could be supplied by the user or generated automatically by the system. In normal ordering practice, the user generates the order number, so this might seem to be more natural. However, it creates a new problem: how to handle a duplicate order number? Solving this would require mechanisms to force users to use unique numbers, or to reject a duplicate. In fact, it is simpler to adopt a more abstract approach that simply requires unique numbers, whether generated by the user or the system (or both, in cooperation).

**Functionality**

Q8. In the context of being able to "change the state of an order" (R0.2), it follows that the system merely inspects the state of current orders and adjusts their status according to the current stock.

Q9. It is said that several orders may mention the same product code (R0.4). This seems an almost unnecessary remark, but it hints that several "simultaneous" orders need to be handled. In this case, how should the stock be allocated to orders? This is not a problem for Case 1 (R1.1). However, in Case 2, the stock is limited by implication (R2.1), so the choice of allocation strategy may lead to different results. For example, the smallest — or the largest — outstanding order for a product might be satisfied first. In the interests of abstractness, it is presumed that orders are satisfied in some "random" manner.[8]

Q10. The system is said to "change the state of an order" (R0.2). Normally, such a system would actually issue an invoice. However, there is no mention of this in the informal problem statement. The conclusion is that the system operates on a set of orders (R1.3), whose status is updated by the system. If an invoice had to be generated, there would be other questions about what it should contain: order reference, product code, quantity, price, etc. However, these matters can be ignored in the case study.

Q11. The system is able to change the state of an order from "pending" to "invoiced" (R0.2). It is not clear whether this means that orders should be explicitly associated with a status. It is presumed so, though the status of an order might be implicit (e.g., because unfulfilled orders are held separately).

Q12. If an order can be fulfilled from stock, its state must be changed to "invoiced" (R0.5). However, nothing is said about the situation where an order cannot be fulfilled because the stock is insufficient. In this situation, the order might be ignored, it might be explicitly rejected, or it might be held until stock becomes available? The first possibility is rather unfriendly and is therefore not considered. As already concluded, the system produces no outputs, so the second possibility is rejected. The third possibility is therefore adopted, and is more consistent with the informal requirements (R0.2, R0.5, R1.3).

Q13. Adopting this third possibility means that, when the system receives a new stock, it must re-examine any unfulfilled orders to see if they can be satisfied. As discussed above, there is then an issue of how stocks should be allocated. Again, a "random" algorithm is assumed.

Q14. The requirements for Case 1 at first appear to be contradictory (R1.3). It is said that there will be no entry flows to the system, yet "the stock and orders are always given [...] in an up-to-date state". Being "given" such information is equivalent to an entry flow. The only interpretation that begins to make sense is that the information is somehow separate from the invoicing function and is updated by some other agency. The system can then consult this information at any time. Presumably, the information is up-to-date only in respect of current stock and order requests. That is, the order status is presumably not up-to-date or the system would be pointless!

Q15. What does cancelling an order mean (R1.2, R2.1)? This suggests an explicit request rather than just omitting an order from the updated list. At what point can an order be cancelled: before it is received by the invoicing system, after reception but before invoicing, after invoicing but before delivery, after delivery? In a real system, these questions would have to be answered concretely. However, as discussed above the purpose of the system seems to be just maintaining a set of current orders. Cancellation must therefore mean removing an order from the pending set. Trying to cancel a non-existent or invoiced order is assumed to be forbidden.

Q16. Is any concurrent or distributed processing required? There is nothing explicit in the requirements, but some implicit possibilities exist. For example, the processing of stock and order updates might be handled concurrently. The invoicing system might also be sub-divided into distributed components. Since these issues are open, a decision should not be forced, though a such decision may be permitted by the specification.

[8] Specifically, the allocation algorithm is not visible to or influenced by the system environment, i.e., it is non-deterministic.

**Limitations**

Q17. By implication (R0.3), an order must carry a product code and a requested quantity. Presumably, the quantity must be a positive integer. Negative quantities might correspond to returned products. A zero-quantity order is conceivable, but it does not seem very useful and should be forbidden. Fractional quantities might be meaningful for products that can be broken down into smaller units, but in the interests of simplicity this was not allowed. Similarly, stock deposits are assumed to be strictly positive integers.

Q18. In Case 1, it is said that all products are in stock (R1.1). This is presumably a hint that stock levels should not be checked before an order is invoiced. However, it is not a realistic assumption, and could even cause the specification to behave inconsistently. It is therefore prudent to check stock levels in this case, even if the check proves to be redundant. (Sometimes it is better if the analyst does not treat literally everything the client says!)

**Non-functional aspects**

Q19. Client requirements would normally include non-functional aspects such as cost, delivery schedule, performance, reliability, integration and testing. However, non-functional aspects can be ignored since the only requirements available are strictly functional.

**Methodology**

Q20. Is Case 1 a simplification or an abstraction of Case 2? Is Case 2 an extension or refinement of Case 1? It is not clear whether consonant requirement specifications are desired. However, it seems sensible to treat the first case as a less detailed form of the second.

## 3.2 Revised Problem Statement

The informal problem statement in section 1 was clarified using the answers in section 3.1. The revised requirements statement used for specification is as follows:

R0. General:

- R0.0 You are to specify a sub-system embedded in a larger warehousing system. You should allow for the possibility of concurrent or distributed processing. Ignore non-functional aspects such as cost, performance, delivery schedule, and testing.
- R0.1 The subject is to support the invoicing of orders. You are not responsible for actually issuing invoices or delivering products.
- R0.2 To invoice is to change the explicit state of an order from "pending" to "invoiced" according to stock levels.
- R0.3 On an order, we have one and only one reference to an ordered product of a certain quantity (a positive integer). The quantity can be different from other orders. Orders carry unique reference numbers that are agreed between the sub-system and the user.
- R0.4 The same product reference can appear in several different orders, some of that may be outstanding at the same time.
- R0.5 The state of the order will be changed to "invoiced" if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product.

R1. Case 1:

- R1.0 You are to treat this as a less detailed form of Case 2.
- R1.1 All the ordered references are intended to be in stock, but you should protect your sub-system against the possibility that this is not actually so.
- R1.2 The stock or the set of the orders may vary due to the entry of new orders or cancelled orders, or due to having a new entry of quantities of products in stock at the warehouse. But we do not have to take these entries into account.
- R1.3 This means that you will not receive two entry flows (orders, entries in stock). The current stock and set of orders are available to you, maintained by another sub-system that informs you when there are new stocks or orders.

R2. Case 2:

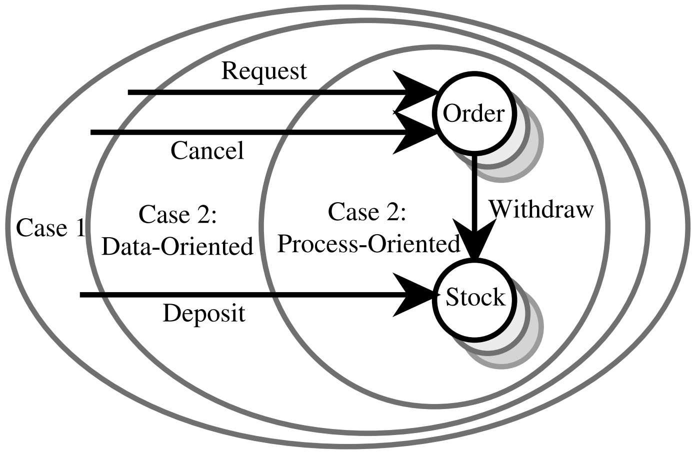


Figure 1: Structure of Specifications

R2.0 You should treat this as the primary case, of which Case 1 is a simplification.

R2.1 You have to take into account the entry of new orders, cancellations of orders, and entries of products in the stock.

R2.2 A cancellation mentions the original order reference. It is forbidden to cancel an order that does not exist or has already been invoiced.

R2.3 Orders that cannot be fulfilled from stock are held until they can be met from new stock. The sequence for satisfying outstanding orders from new stock is at your discretion.

# 4 The LOTOS and E-LOTOS Specifications

The case study is mainly data-oriented, since it describes a database. For this reason, the corresponding LOTOS specification makes significant use of data types. However, there is a modelling choice to be made of whether to represent stocks and orders as processes or as data values. For this reason, *two* specification approaches are presented later in the paper. These give some idea of the range of styles open to the LOTOS specifier.

Since E-LOTOS is an improved version of LOTOS, the authors felt it would be interesting to see how its specification style(s) differed from LOTOS. E-LOTOS specifications are presented first, followed by specifications in current LOTOS. Since the languages differ, each has been written in native style and the specifications are not just syntactic translations of each other. Each portion of a formal specification is preceded by an informal explanation. In the specifications that follow, the authors have used their own convention for the case of identifiers (keywords in bold font, variables in lower case, other identifiers with an initial capital).

As argued in section 3.1, Case 1 is just an abstraction of Case 2. For Case 2, the process-oriented style introduces some internal structure to the specification. The structure of the specifications to be presented can be pictured as in figure 1. Case 1 has no inputs, and thus has no externally observable behaviour. The inputs in Case 2 are *Request* (place an order), *Cancel* (remove an order) and *Deposit* (supply new stock). The process-oriented version of Case 2 introduces an internal communication *Withdraw* (satisfy an order from stock).

The paper describes {Case 2, Case 1}×{process-oriented, data-oriented}×{E-LOTOS, LOTOS}, i.e., eight specifications in total. However, as will be seen there is considerable commonality in the various approaches.

## 4.1 Process-Oriented E-LOTOS Specification

The process-oriented specification of the invoicing system might be regarded as object-based. Orders and stock are individual objects that encapsulate an identity (order reference or product code), state (order or stock status) and services (request order, deposit stock, etc.). The identity of an order or stock item allows it, out of the whole collection, to synchronise on messages intended for it.

Figure 2 shows the architecture of the process-oriented (E-LOTOS and LOTOS) specifications for Case 2. Note that, E-LOTOS (like LOTOS) models a system as a collection of communicating processes. The communication ports of a process are called *gates* that, in E-LOTOS, are typed, in order to allow static checking of the kinds of values that are communicated. Processes are parameterised by their gates and state variables. They make event offers such as *Withdraw(!prd,?newamt)* that may be synchronised (matched) at a gate with their environment. Synchronised offers become actual events. A fixed value in an event offer is preceded by "!", whereas an open value to be determined in an event offer is preceded by "?".

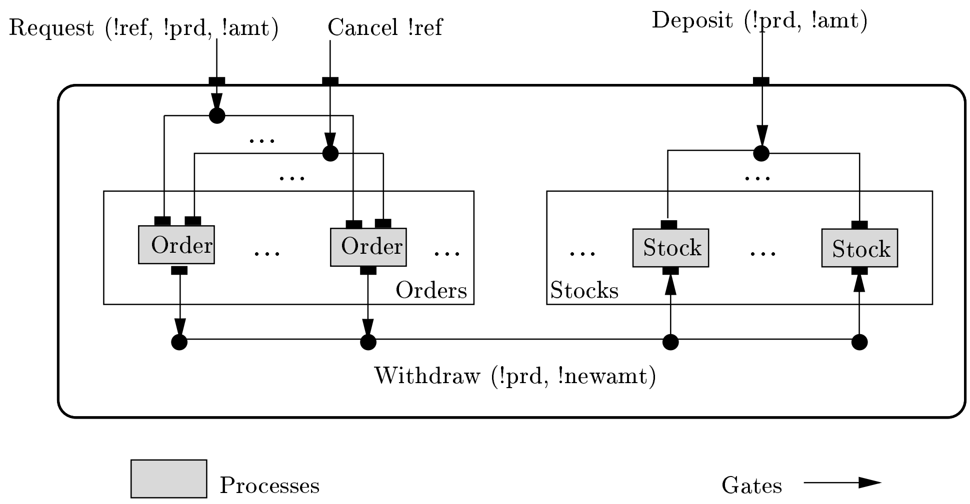


Figure 2: Architecture of the Process-Oriented Description

For convenience, the data types and processes are specified here in a separate module. In E-LOTOS, data values domains are described by constructed types, and operations on these values are described by functions.

For clarity, separate types are introduced for order references, product codes and product amounts. For simplicity, these simply rename the natural number type; library types like this can be used without explicit importing. If desired, structured types could be introduced later for order references and product codes.

**module** OrderStock **is**
  **type** Reference **renames** Nat **end type**
  **type** Product **renames** Nat **end type**
  **type** Amount **renames** Nat **end type**

The status of an order is defined using an enumerated type; *None* is used for an order that is not current. The type of the *Request* gate is a record containing the reference of the order, the product, and the amount fields. The type of the *Deposit* gate is a record containing the product stocked and the amount fields.

  **type** Status **is enum** None, Pending, Invoiced **end type**
  **type** Request **is record** Ref:Reference, Prod:Product, Amt:Amount **end type**
  **type** Stock **is record** Prod:Product, Amt:Amount **end type**

An *Order* object repeatedly accepts order requests from the environment, accepts order cancellations from the environment, and makes withdrawals from stock objects. A choice ([]) is made from these possibilities. A new order is permitted ([*condition*] after event offer) only if the reference is unused (status *None*) and the amount is positive; the order then becomes pending. Cancellation is allowed only if the order is pending, at which point it ceases to be used. A pending order may ask for withdrawal of stock. The stock object with the corresponding product will synchronise on this offer (*Withdraw(!prd, !amt)*) if there is sufficient stock. If the order cannot be currently satisfied, the withdrawal request remains open until sufficient stock exists. At this point the order becomes invoiced.

```
process Order [Request:Request, Cancel:Reference, Withdraw:Stock]
 (ref:Reference, prd:Product, amt:Amount, sta:Status) is
  loop
    Request(!ref, ?prd, ?amt) [(sta == None) and (amt > 0)];
    sta := Pending
  []
    Cancel(!ref) [sta == Pending];
    sta := None
  []
    Withdraw(!prd, !amt) [sta == Pending];
    sta := Invoiced
  end loop
end process
```

A *Stock* object repeatedly accepts deposits from the environment and withdrawals from order objects. New stock (of positive amount) is simply added to the current stock-holding. Withdrawal is permitted if the requested amount can be taken from the current stock. Note that several orders may compete simultaneously for withdrawal of the same stock. Since these orders are handled concurrently, the sequence in which they are satisfied is non-deterministic.

```
process Stock [Deposit, Withdraw:Stock] (prd:Product, amt:Amount) is
  var newamt:Amount in
    loop
      Deposit(!prd, ?newamt) [newamt > 0];
      amt := amt + newamt
    []
      Withdraw(!prd, ?newamt) [newamt <= amt];
      amt := amt − newamt
    end loop
  end var
end process
```

The module concludes by defining infinite sets of order and stock processes, each independently in parallel (|||). These are obtained by explicit recursion over the order reference and stock product code. An order is initialised with its reference and “not in use” status. A stock item is initialised with its product code and a zero amount.

```
  process Orders [Request:Request, Cancel:Reference, Withdraw:Stock] (ref:Reference) is
    Order [Request, Cancel, Withdraw] (ref, 0, 0, None)
  |||
    Orders [Request, Cancel, Withdraw] (ref + 1)
  end process
  process Stocks [Deposit, Withdraw:(Product,Amount)] (prd:Product) is
    Stock [Deposit, Withdraw] (prd, 0)
  |||
    Stocks [Deposit, Withdraw] (prd + 1)
  end process
end module
```

The overall system for the Case 2 imports the module for orders and stocks. The communication gates (all inputs in this case) are introduced, and the lists of values they carry are specified.

```
specification Invoicing import OrderStock is
  gates Request:Request, Cancel:Reference, Deposit:Stock
  behaviour
    hide Withdraw:Stock in
      Orders [Request, Cancel, Withdraw] (0)
    |[Withdraw]|
      Stocks [Deposit, Withdraw] (0)
    end hide
end specification
```

Communication between orders and stocks is via an internal gate *Withdraw*. The order and stock processes synchronise on withdrawal (|[*Withdraw*]|). As new orders and stocks arrive, the processes will update their state and will communicate to satisfy orders.

Following the discussion in section 3.1, the Case 1 is viewed as an abstraction of the Case 2. Specifically, the gates for communicating with the system are treated as hidden and thus there is no externally observable behaviour. This affects the top-level behaviour as detailed in the following; only the changes relative to Case 2 are given.

```
specification Invoicing import OrderStock is
  behaviour
    hide Request:Request, Cancel:Reference, Deposit, Withdraw:Stock in
      Orders [Request, Cancel, Withdraw] (0)
    |[Withdraw]|
      Stocks [Deposit, Withdraw] (0)
    end hide
end specification
```

## 4.2 Process-Oriented LOTOS Specification

The equivalent process-oriented specification in LOTOS is similar, except that modules are not available. The system is a complete specification that is non-terminating (**noexit**). Natural numbers are selected from the standard library, and a subtract operation (not in the library) is introduced. This is given in a new type that imports and extends the standard natural number type. An operation is declared by giving its signature: *parameters* → *result*. Equations are grouped under **ofsort** according to the return value of the operations being specified. When a variable may take any value in an equation, it is declared by **forall**. "−" is an infix operation that takes two naturals and returns a natural. It is defined by characteristing equations that use the successor operation in the library for producing consecutive naturals. Since a natural is non-negative, subtraction cannot lead to less than 0. Equations are usually straightforward but tedious to write. Each distinct form of an operation's parameters leads to a separate equation. For this type, the forms are zero and successor of some number.

```
specification Invoicing [Request, Cancel, Deposit] : noexit
  library NaturalNumber endlib
  type Natural is NaturalNumber
    opns _ − _ : Nat, Nat -> Nat
    eqns forall n1,n2:Nat
      ofsort Nat
        0 − n2 = 0;
        n1 − 0 = n1;
        Succ(n1) − Succ(n2) = n1 − n2;
  endtype
```

Order references, product codes, and product amounts are again specified by renaming naturals. The status type is much as for E-LOTOS.

```
  type Reference is Natural renamedby
    sortnames Reference for Nat
  endtype
  type Product is Natural renamedby
    sortnames Product for Nat
  endtype
  type Amount is Natural renamedby
    sortnames Amount for Nat
  endtype
  type Status is
    sorts Status
    opns None, Pending, Invoiced : -> Status
  endtype
```

The overall behaviour for the Case 2 is similar to the corresponding E-LOTOS case.

```
  behaviour
    hide Withdraw in
      Orders [Request, Cancel, Withdraw] (0 of Reference)
    |[Withdraw]|
```

```
    Stocks [Deposit, Withdraw] (0 of Product)
where
process Orders [Request, Cancel, Withdraw] (ref:Reference) : noexit :=
  Order [Request, Cancel, Withdraw] (ref, 0 of Product, 0 of Amount, None)
|||
  Orders [Request, Cancel, Withdraw] (Succ(ref))
endproc
process Stocks [Deposit, Withdraw] (prd:Product) : noexit :=
  Stock [Deposit, Withdraw] (prd, 0 of Amount)
|||
  Stocks [Deposit, Withdraw] (Succ(prd))
endproc
```

Except for syntactic differences, the *Order* and *Stock* objects are similar to their E-LOTOS counterparts. Loops must be achieved by explicit recursion in LOTOS.

```
process Order [Request, Cancel, Withdraw]
 (ref:Reference, prd:Product, amt:Amount, sta:Status) : noexit :=
  [sta = None] ->
    Request !ref ?prd:Product ?amt:Amount [amt gt 0];
    Order [Request, Cancel, Withdraw] (ref, prd, amt, Pending)
[]
  [sta = Pending] ->
    (
      Cancel !ref;
      Order [Request, Cancel, Withdraw] (ref, 0 of Product, 0 of Amount, None)
    []
      Withdraw !prd !amt;
      Order [Request, Cancel, Withdraw] (ref, prd, amt, Invoiced)
    )
endproc
process Stock [Deposit, Withdraw] (prd:Product, amt:Amount) : noexit :=
  Deposit !prd ?newamt:Amount [newamt gt 0];
  Stock [Deposit, Withdraw] (prd, amt + newamt)
[]
  Withdraw !prd ?newamt:Amount [newamt le amt];
  Stock [Deposit, Withdraw] (prd, amt - newamt)
endproc
endspec
```

The specification for Case 1 is obtained, as in E-LOTOS, by hiding the gates for communicating with the system. This affects the top-level behaviour as detailed in the following; only the changes relative to Case 2 are given.

```
specification Invoicing : noexit
  ...
  behaviour
    hide Request, Cancel, Deposit, Withdraw in
      Orders [Request, Cancel, Withdraw] (0)
    |[Withdraw]|
      Stocks [Deposit, Withdraw] (0)
endspec
```

## 4.3 Data-Oriented E-LOTOS Specification

In this approach, orders and stocks are defined by data values rather than processes. The system is described as a single process accepting requests, cancellations, and deposits. This process uses functions operating on data types representing the invoiced orders and the existing stocks. Figure 3 shows the architecture of the E-LOTOS and LOTOS data-oriented descriptions.

To describe data, a data type module is used, similar to that for the process-oriented version. The complete type for an order is given as a record containing reference, product, amount, and status fields. A collection of orders is described by a list. A collection of stocks is similar, but the element of the list

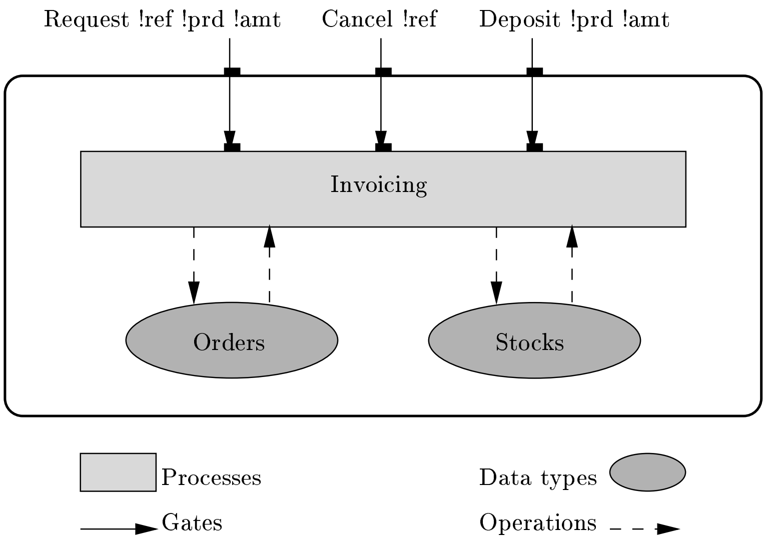


Figure 3: Architecture of the Data-Oriented Description

is a stock[9]. Note that list type declaration generates automatically the classical operations on lists like *nil*, *cons*, *head*, etc.

```
module OrderStock is
  type Reference renames Nat end type
  type Product renames Nat end type
  type Amount renames Nat end type
  type Status is enum None, Pending, Invoiced end type
  type Request is record Ref:Reference, Prod:Product, Amt:Amount end type
  type Stock is record Prod:Product, Amt:Amount end type
  type Order is record Ref:Reference, Prod:Product, Amt:Amount, Stat:Status end type
  type Stocks is list of Stock end type
  type Orders is list of Order end type
```

An order may be removed from a collection using the *RemOrder* function. This function is described using the imperative facilities offered by the variant of E-LOTOS used, i.e., LOTOS NT [GS98]. The parameter *ords* is modified in place so it is declared "**inout**". A field of a record is selected using the "." operator, for example, *ord1.Ref* selects the field *Ref* of the *Order* record. *StatOrder* is introduced to retrieve the status of an order in the collection. This function is described in a functional style, by using recursive calls.

```
function RemOrder (ref:Reference, inout ords:Orders) is
  var newords:Orders:=nil, ord1:Order in
    while ords != nil do
      ord1 := head (ords);
      ords := tail (ords);
      if ord1.Ref == ref then
        break
      else
        newords := append (newords, cons (ord1, nil))
      end if
    end while;
  ords := append (newords, ords)
  end var
end function
```

[9]Stocks of the same product need to be amalgamated, so stock is not strictly a set.

```
function StatOrder (ref:Reference, ords:Orders) : Status is
  if ords == nil then
    return None
  elsif (head (ords)).Ref == ref then
    return (head (ords)).Stat
  else
    return StatOrder (ref, tail (ords))
  end if
end function
```

A stock may be added or removed from a collection using respectively *AddStock* and *RemStock* procedures. The assignment of a record field is described using the update operator on records, for example, the expression *stk1.{Amt => stk1.Amt + stk.Amt}* denotes the actualisation of the field *Amt* with the value given by *stk1.Amt + stk.Amt*. The *StockOf* function is introduced to retrieve the amount of a product in the collection of stocks.

```
function AddStock (stk:Stock, inout stks:Stocks) is
  var newstks:Stocks:=nil, stk1:Stock, found:Bool:=false in
    while stks != nil do
      stk1 := head (stks);
      stks := tail (stks);
      if stk1.Prod == stk.Prod then
        newstks := cons (stk1.{Amt => stk1.Amt + stk.Amt}, newstks);
        break
      else
        newstks := cons (stk1, newstks)
      end if
    end while;
    if not (found) then newstks := cons (stk, newstks) end if;
    stks := append (newstks, stks)
  end var
end function

function RemStock (stk:Stock, inout stks:Stocks) is
  var newstks:Stocks:=nil, stk1:Stock in
    while stks != nil do
      stk1 := head (stks);
      stks := tail (stks);
      if stk1.Prod == stk.Prod then
        newstks := cons (stk1.{Amt => stk1.Amt - stk.Amt}, newstks);
        break
      else
        newstks := cons (stk1, newstks)
      end if
    end while;
    stks := append (newstks, stks)
  end var
end function

function StockOf (prd:Product, stks:Stocks) : Amount is
  if stks == nil then
    return 0
  elsif (head (stks)).Prod == prd then
    return (head (stks)).Amt
  else
    return StockOf (prd, tail (stks))
  end if
end function
```

Invoicing orders is carried out by a procedure that takes current orders and stocks. Each order is checked in a loop, from first order in the list to the last. Orders are thus fulfilled in a FIFO (first-in first-out) order, and not non-deterministically as in the process-oriented version. Non-determinism could have been achieved, but by complicating the specification. The product code of the current order is used to extract the stock level. If the order is pending and there is sufficient stock, the order is marked as invoiced and the stock level is updated. After all orders have been processed, the function exits with updated values of orders and stocks. If an order cannot be fulfilled, it may be satisfied later when invoicing is repeated on receipt of new stock. The call of the procedure *RemStock* is prefixed by the keyword "**eval**", and the "**inout**" actual parameter *stcks* is prefixed by "?".

```
    function Invoice(inout ords:Orders, inout stks:Stocks) is
      var newords:Orders:=nil, ord:Order in
        while ords != nil do
          ord := head (ords);
          if (ord.Stat == Pending) and (StockOf (ord.Prod, stks) >= ord.Amt) then
            newords := cons (ord.{Stat=>Invoiced}, newords);
            eval RemStock (Stock (ord.Prod, ord.Amt), ?stks)
          else
            newords := cons (ord, newords);
          end if
          ords := tail (ords);
        end while
        ords := newords;
      end var
    end function
  end module
```

The system specification for the Case 2 is like that for the corresponding process-oriented version except that local variables are introduced. In particular, orders and stocks are initialised as empty. The main behaviour repeatedly accepts order requests, order cancellations, and stock deposits. The logic is as already seen, except that existence of an order is checked against the *Orders* list. Each branch of the loop updates orders or stocks as appropriate. The *Invoice* function is then called to schedule orders and alter stocks.

```
  specification Invoicing import OrderStock is
    gates Request:TypeReference, Cancel:Reference, Deposit:Stock
    behaviour
      var ords:Orders := nil, stks:Stocks := nil, ref:Reference, prd:Product, amt:Amount in
        loop
          (
            Request(?ref, ?prd, ?amt) [(StatOrder (ref, ords) == None) and (amt > 0)];
            ords := cons (Order(ref, prd, amt, Pending), ords)
          []
            Cancel ?ref [StatOrder (ref, ords) == Pending];
            eval RemOrder (ref, ?ords)
          []
            Deposit(?prd, ?amt) [amt > 0];
            eval AddStock (Stock (prd, amt), ?stks)
          );
          eval Invoice (?ords, ?stks)
        end loop
      end var
  end specification
```

The specification for Case 1 is obtained by hiding the gates for communicating with the system. This affects the top-level behaviour as detailed in the following; only the changes relative to Case 2 are given.

```
  specification Invoicing import OrderStock is
    behaviour
      hide Request:Request, Cancel:Reference, Deposit:Stock in
```

## 4.4 Data-Oriented LOTOS Specification

The specification begins in much the same way as the process-oriented LOTOS version, except that boolean equality for status values has to be defined. Boolean equality is defined for two status values so that compound boolean expressions involving status can be written. Following normal LOTOS practice, equality is defined using an auxiliary function *Ord* (for ordinal) that maps values to the natural numbers.

```
specification Invoicing [Request, Cancel, Deposit] : noexit
  library Boolean, NaturalNumber endlib
  type Natural is NaturalNumber
    opns _ − _ : Nat, Nat → Nat
    eqns forall n1,n2:Nat
      ofsort Nat
        0 − n2 = 0;
        n1 − 0 = n1;
        Succ(n1) − Succ(n2) = n1 − n2;
  endtype
  type Reference is Natural renamedby
    sortnames Reference for Nat
  endtype
  type Product is Natural renamedby
    sortnames Product for Nat
  endtype
  type Amount is Natural renamedby
    sortnames Amount for Nat
  endtype
  type Status is Boolean, NaturalNumber
    sorts Status
    opns
      None, Pending, Invoiced : → Status
      Ord : Status → Nat
      _ eq _ : Status, Status → Bool
    eqns forall sta1,sta2:Status
      ofsort Nat
        Ord(None) = 0;
        Ord(Pending) = Succ(0);
        Ord(Invoiced) = Succ(Succ(0));
      ofsort Bool
        sta1 eq sta2 = Ord(sta1) eq Ord(sta2);
  endtype
```

The reference, product, amount and status types are imported as components of an order. Stock is built from product and amount types. Since LOTOS does not have a record construct, "make record" operations are needed.

```
  type Order is Reference, Product, Amount, Status
    sorts Order
    opns MkOrder : Reference, Product, Amount, Status → Order
  endtype
  type Stock is Product, Amount
    sorts Stock
    opns MkStock : Product, Amount → Stock
  endtype
```

Orders and stocks might have been defined using the generic set type in the library. However, orders and stocks have been specified from scratch since sets are not entirely appropriate[10]. *NoOrders* is an empty collection of orders. An order may be added to or removed from this using the *AddOrder* and *RemOrder* operations. *StatOrder* is introduced to retrieve the status of an order in the collection. Each operation is defined by equations as already seen. In this case, the distinct forms of operation parameter to be considered are a collection with no orders and with at least one order. Conditional equations (*condition* ⇒ *equation*) apply only if the condition holds.

```
  type Orders is Order, Status
    sorts Orders
    opns
      NoOrders : → Orders
      AddOrder : Order, Orders → Orders
```

[10] Stocks of the same product need to be amalgamated, so stock is not strictly a set. Identical orders should be allowed, so a bag rather than a set is needed.

```
      RemOrder : Order, Orders -> Orders
      StatOrder : Reference, Orders -> Status
    eqns
      forall ref1,ref2:Reference, prd1,prd2:Product, amt1,amt2:Amount,
       sta1,sta2:Status, ords:Orders
        ofsort Status
          StatOrder(ref1, NoOrders) = None;
          ref1 eq ref2 =>
            StatOrder(ref1, AddOrder(MkOrder(ref2, prd2, amt2, sta2), ords)) = sta2;
          ref1 ne ref2 =>
            StatOrder(ref1, AddOrder(MkOrder(ref2, prd2, amt2, sta2), ords)) = StatOrder(ref1, ords);
        ofsort Orders
          ref1 eq ref2 =>
            RemOrder(MkOrder(ref1, prd1, amt1, sta1),
              AddOrder(MkOrder(ref2, prd2, amt2, sta2), ords)) = ords;
          ref1 ne ref2 =>
            RemOrder(MkOrder(ref1, prd1, amt1, sta1),
              AddOrder(MkOrder(ref2, prd2, amt2, sta2), ords)) =
                AddOrder(MkOrder(ref2, prd2, amt2, sta2),
                  RemOrder(MkOrder(ref1, prd1, amt1, sta1), ords));
  endtype
```

A stock collection is defined in a similar way. The operations particular to stocks are *InStock* (to check if a product is stocked) and *StockOf* (to check the stock level).

```
  type Stocks is Stock
    sorts Stocks
    opns
      NoStocks : -> Stocks
      AddStock : Stock, Stocks -> Stocks
      RemStock : Stock, Stocks -> Stocks
      InStock : Product, Stocks -> Bool
      StockOf : Product, Stocks -> Amount
    eqns
      forall prd1,prd2:Product, amt1,amt2:Amount, stks:Stocks
        ofsort Bool
          InStock(prd1, NoStocks) = false;
          InStock(prd1, AddStock(MkStock(prd2, amt2), stks)) =
            (prd1 eq prd2) or InStock(prd1, stks);
        ofsort Stocks
          prd1 eq prd2 =>
            AddStock(MkStock(prd1, amt1), AddStock(MkStock(prd2, amt2), stks)) =
              AddStock(MkStock(prd1, amt2 + amt1), stks);
          (prd1 ne prd2) and InStock(prd1, stks) =>
            AddStock(MkStock(prd1, amt1), AddStock(MkStock(prd2, amt2), stks)) =
              AddStock(MkStock(prd2, amt2), AddStock(MkStock(prd1, amt1), stks));
        ofsort Stocks
          RemStock(MkStock(prd1, amt1), NoStocks) = NoStocks;
          prd1 eq prd2 =>
            RemStock(MkStock(prd1, amt1), AddStock(MkStock(prd2, amt2), stks)) =
              AddStock(MkStock(prd1, amt2 - amt1), stks);
          prd1 ne prd2 =>
            RemStock(MkStock(prd1, amt1), AddStock(MkStock(prd2, amt2), stks)) =
              AddStock(MkStock(prd2, amt2), RemStock(MkStock(prd1, amt1), stks));
        ofsort Amount
          StockOf(prd1, NoStocks) = 0;
          prd1 eq prd2 =>
            StockOf(prd1, AddStock(MkStock(prd2, amt2), stks)) = amt2;
          prd1 ne prd2 =>
            StockOf(prd1, AddStock(MkStock(prd2, amt2), stks)) = StockOf(prd1, stks);
  endtype
```

Since a LOTOS operation can return only one result (unless result types are grouped in a composite type), invoicing is computed by separate operations: *UpdateOrders* and *UpdateStocks*. In both cases, the collection of orders is processed one by one. (Like the data-oriented E-LOTOS specification, this means that order fulfillment is deterministic, but not in the fixed order of reference numbers.) If an order is pending and the stocks are sufficient for the requested amount, the order status is set to *Invoiced* and the stock level is updated.

```
type Updates is Orders, Stocks
  opns
    UpdateOrders : Orders, Stocks -> Orders
    UpdateStocks : Orders, Stocks -> Stocks
  eqns
    forall ref:Reference, prd:Product, amt:Amount, sta:Status, ords:Orders, stks:Stocks
      ofsort Orders
        UpdateOrders(NoOrders, stks) = NoOrders;
        (sta eq Pending) and (StockOf(prd, stks) ge amt) =>
          UpdateOrders(AddOrder(MkOrder(ref, prd, amt, sta), ords), stks) =
            AddOrder(MkOrder(ref, prd, amt, Invoiced),
             UpdateOrders(ords, RemStock(MkStock(prd, amt), stks)));
        (sta eq Invoiced) or (StockOf(prd, stks) lt amt) =>
          UpdateOrders(AddOrder(MkOrder(ref, prd, amt, sta), ords), stks) =
            AddOrder(MkOrder(ref, prd, amt, sta), UpdateOrders(ords, stks));
      ofsort Stocks
        UpdateStocks(NoOrders, stks) = stks;
        (sta eq Pending) and (StockOf(prd, stks) ge amt) =>
          UpdateStocks(AddOrder(MkOrder(ref, prd, amt, sta), ords), stks) =
            UpdateStocks(ords, RemStock(MkStock(prd, amt), stks));
        (sta eq Invoiced) or (StockOf(prd, stks) lt amt) =>
          UpdateStocks(AddOrder(MkOrder(ref, prd, amt, sta), ords), stks) =
            UpdateStocks(ords, stks);
endtype
```

Now the dynamic behaviour of the system for the Case 2 is given as a call of the *Invoice* process. This takes the same gates as the overall system and starts out with empty orders and stocks.

```
behaviour Invoice [Request, Cancel, Deposit] (NoOrders, NoStocks) where
```

The specification is similar to that for E-LOTOS, though the syntax is different. Again, explicit recursion must be used to express a loop. Each branch of the choice produces an updated pair of order-stock values. These are exported to the final calculation in a construct of the form: **exit** *(orders,stocks)* ≫ **accept** *neworders,newstocks* **in**. This is called *enabling*, and allows a terminating behaviour to export its results to another behaviour. The recursive call to *Invoice* updates the orders and stocks following invoicing.

```
process Invoice [Request, Cancel, Deposit] (ords:Orders, stks:Stocks) : noexit :=
 (
    Request ?ref:Reference ?prd:Product ?amt:Amount [(StatOrder(ref,ords) eq None) and (amt gt 0)];
    exit (AddOrder(MkOrder(ref, prd, amt, Pending), ords), stks)
  []
    Cancel ?ref:Reference [StatOrder(ref, ords) eq Pending];
    exit (RemOrder(MkOrder(ref, 0, 0, Pending), ords), stks)
  []
    Deposit ?prd:Product ?amt:Amount [amt gt 0];
    exit (ords, AddStock(MkStock(prd, amt), stks))
 )
>>
  accept neword:Orders, newstks:Stocks in
    Invoice [Request, Cancel, Deposit]
     (UpdateOrders(newords, newstks), UpdateStocks(newords, newstks))
endproc
endspec
```

The specification for Case 1 is obtained by hiding the gates for communicating with the system. This affects the top-level behaviour as detailed in the following; only the changes relative to Case 2 are given.

**specification** Invoicing : **noexit**
...
**behaviour**
**hide** Request, Cancel, Deposit **in**
Invoice [Request, Cancel, Deposit] (NoOrders, NoStocks) **where**

# 5 On the Formal Validation of Specifications

Since E-LOTOS is currently under balloting within ISO, the tools for this language are not yet sufficiently mature to provide a mean for verification of the E-LOTOS descriptions. The E-LOTOS specifications should hence be regarded as conceptual at this stage, although they have been statically checked using the TRAIAN tool [Viv97].

As regards the LOTOS specifications, a first analysis by testing has been undertaken using the LITE[11] tool [Eij91], in a form of white-box testing:

- The data type definitions were checked by evaluating operations on test values conforming to each distinct form of parameter.
- The behavioural specifications were checked using scenarios that exercise each significant case.
  - For order requests the scenarios included duplicated references, zero amounts, products not currently in stock, amounts less than the current stock, amounts exactly equal to the current stock, and multiple orders for the same product.
  - For order cancellations the scenarios dealt with non-existent references, pending and invoiced orders.
  - For stock deposits the scenarios included new product codes, existing product codes, zero amounts, and stocks for pending orders.

The analysis was documented by giving the scenarios and the reactions of the system to them. Normally, the client would be involved in confirming the completeness and correctness of testing, but that was not possible for this case study. However, testing cannot ensure the correctness of the specification and the equivalence between the various specifications. Such aims need the analyse of the state space of the specification by formal verification.

To perform formal verification, we used *model-checking*, which proceeds autonomously but is only applicable to systems with a finite state space. We consider for this the CADP toolbox [FGK+96], which provides state-of-the-art verification features.

The CADP[12] toolbox is dedicated to the design and the verification of communication protocols and distributed systems. Initiated in 1986, it offers an integrated set of functionalities ranging from interactive simulation to exhaustive, model-based verification methods, and currently includes sophisticated approaches to deal with large case studies. In addition to LOTOS, it also supports lower level formalisms such as finite state machines and networks of communicating automata.

In the sequel, we briefly present only the tools of the CADP toolbox used throughout this case study:

- CÆSAR [GS90] and CÆSAR.ADT [Gar89] are compilers that translate a LOTOS program into a Labelled Transition System (LTS for short) describing its exhaustive behaviour. This LTS can be represented either *explicitly*, as a set of states and transitions, or *implicitly*, as a library of C functions allowing to execute the program behaviour in a controlled way.
- ALDÉBARAN [FKM93] is a verification tool for comparing or minimizing LTSs with respect to (bi)simulation relations [Par81, Mil89]. Initially designed to deal with explicit LTSs produced by CÆSAR, it has been extended to also handle networks of communicating automata (for on-the-fly and symbolic verification). Several simulation and bisimulation relations are implemented within ALDÉBARAN, which offers a wide spectrum for expressing such behavioural specifications.

[11] LOTOS Integrated Tool Environment
[12] CÆSAR/ALDÉBARAN Development Package

- XTL (*eXecutable Temporal Language*) [Mat98, MG98] is a functional-like programming language designed to allow an easy, compact implementation of various temporal logic operators. Besides the usual predefined types (booleans, integers, etc.) the XTL language defines special types, such as sets of states, transitions, and labels of the LTS. It offers primitives to access the information contained in states and labels, to obtain the initial state, and to compute the successors and predecessors of states and transitions. The temporal operators can be easily implemented using these functions together with recursive user-defined functions working with sets of states and/or transitions of the LTS. A prototype compiler for XTL has been developed, and several temporal logics like HML [HM85], CTL [EH86], ACTL [NV90], and LTAC [QS83] have been easily implemented in XTL.

# 6 Model Generation

In order to perform verification of the Invoicing case-study by model-checking, we first generate, using the CADP tools, various LTSs corresponding to the LOTOS descriptions proposed in section 4.

First, we give the formal definition of the LTS model. Then, we present our methodology and the experimental results concerning the model generation.

## 6.1 The LTS Model

According to the operational semantics of LOTOS, LOTOS programs can be translated into (possibly infinite) LTSs, which encode all their possible execution sequences. An LTS is formally defined as a quadruple $M = \langle Q, A, T, q_{init} \rangle$, where:

- $Q$ is the set of *states* of the program;
- $A$ is the set of *actions* performed by the program. An action $a \in A$ is a tuple $G\ V_1, ..., V_n$ where $G$ is a *gate* and $V_1$, ..., $V_n$ ($n \geq 0$) are the values exchanged (i.e., sent or received) during the rendezvous at $G$. For the *silent* action $\tau$, the value list must be empty ($n = 0$);
- $T \subseteq Q \times A \times Q$ is the *transition relation*. A transition $\langle q_1, a, q_2 \rangle \in T$ (written also "$q_1 \xrightarrow{a} q_2$") means that the program can move from state $q_1$ to state $q_2$ by performing action $a$;
- $q_{init} \in Q$ is the *initial state* of the program.

## 6.2 Methodology

To be able to generate the LTS of a given description, this LTS must be finite and have a tractable size. The LOTOS descriptions given in the first part of this report cannot be used directly to generate finite LTSs with CADP. We present below the main reasons and the alternative solutions we consider.

- In order to provide an efficient compilation of the data operations, the CÆSAR.ADT compiler needs to know the operations which are the constructors of each sort. Also, the ACTONE equations are considered as rewriting rules (oriented from left to right), and equations between constructors are not allowed. For this reason, we identify the constructors for each declared sort and then we transform (as described in [Gar89]) some of the equations given in section 4.4 in order to eliminate equations between constructors. For example, in the following declaration:

```
type Orders is Order, Status
  sorts Orders
  opns
    NoOrders (*! constructor *) : -> Orders
    AddOrder (*! constructor *) : Order, Orders -> Orders
    RemOrder : Order, Orders -> Orders
    StatOrder : Reference, Orders -> Status
  eqns
   ...
endtype
```

we identify (by the special comment *(*! constructor *)*) the operations *NoOrders* and *AddOrder* as being the constructors of the sort *Orders*, because there are no equations describing the rewriting of terms containing these operations. This is not so simple for the following declaration:

```
type Stocks is Stock
  sorts Stocks
  opns
    NoStocks : -> Stocks
    AddStock : Stock, Stocks -> Stocks
    RemStock : Stock, Stocks -> Stocks
    InStock : Product, Stocks -> Bool
    StockOf : Product, Stocks -> Amount
  eqns
   ...
      ofsort Stocks
        prd1 eq prd2 =>
          AddStock(MkStock(prd1, amt1), AddStock(MkStock(prd2, amt2), stks)) =
            AddStock(MkStock(prd1, amt2 + amt1), stks);
        (prd1 ne prd2) and InStock(prd1, stks) =>
          AddStock(MkStock(prd1, amt1), AddStock(MkStock(prd2, amt2), stks)) =
            AddStock(MkStock(prd2, amt2), AddStock(MkStock(prd1, amt1), stks));
   ...
endtype
```

where the operation *AddStock* cannot be identified as a constructor because there exists two equations describing the rewriting of terms containing the *AddStock* operation. For this reason, we introduce a new operator *AddStock0* with the same profile as *AddStock*, we mark it as constructor, and substitute *AddStock* with *AddStock0* in all the places where *AddStock* is used as a constructor. A new equation for the *AddStock* operation is added in order to describe its behaviour if the list of stocks is empty[13]. With these modifications, the example above becomes:

```
type Stocks is Stock
  sorts Stocks
  opns
    NoStocks (*! constructor *) : -> Stocks
    AddStock0 (*! constructor *) : Stock, Stocks -> Stocks
    AddStock : Stock, Stocks -> Stocks
    RemStock : Stock, Stocks -> Stocks
    InStock : Product, Stocks -> Bool
    StockOf : Product, Stocks -> Amount
  eqns
   ...
      ofsort Stocks
        prd1 eq prd2 =>
          AddStock(MkStock(prd1, amt1), AddStock0(MkStock(prd2, amt2), stks)) =
            AddStock0(MkStock(prd1, amt2 + amt1), stks);
        (prd1 ne prd2) and InStock(prd1, stks) =>
          AddStock(MkStock(prd1, amt1), AddStock0(MkStock(prd2, amt2), stks)) =
            AddStock0(MkStock(prd2, amt2), AddStock(MkStock(prd1, amt1), stks));
        AddStock(MkStock(prd1, amt1), NoStocks) = MkStock(prd1, amt1);
   ...
endtype
```

- Since we use model-checking techniques to validate the LOTOS descriptions, these descriptions should generate a finite model. This is not the case for the descriptions given in section 4.4, since they use (unbounded) natural numbers. To generate finite LTSs, we restricted to finite sets the domains of all description parameters. Therefore, in each experiment we fixed a maximal value for the *Reference*, *Product*, and *Amount* sorts which are noted respectively $M_r$, $M_p$, and $M_a$.

[13] This is not needed if the algebraic semantics of ACTONE is used, but in the interpretation of equations as rewriting rules its absence leads to an incomplete specification.

- Another source for infinite state space is the use of recursion in the parallel composition operator. For example, the following process declaration allows to express that an infinite number of *Orders* can be generated:

  **process** Orders [Request, Cancel, Withdraw] (ref:Reference) : **noexit** :=
  Order [Request, Cancel, Withdraw] (ref, 0 **of** Product, 0 **of** Amount, None)
  |||
  Orders [Request, Cancel, Withdraw] (Succ(ref))
  **endproc**

  To generate finite LTSs, we restricted the number of *Order* processes by explicitly instantiating them. For example, if the maximal number of orders is two (i.e., $M_r = 2$), the declaration above becomes:

  **process** Orders [Request, Cancel, Withdraw] (ref:Reference) : **noexit** :=
  Order [Request, Cancel, Withdraw] (0, 0 **of** Product, 0 **of** Amount, None)
  |||
  Order [Request, Cancel, Withdraw] (Succ(0) **of** Product, 0 **of** Product, 0 **of** Amount, None)
  **endproc**

Despite these restrictions, the state space of the system quickly becomes intractable. For this reason, we consider only certain cases of the LOTOS descriptions. These cases are given by the following product:

$$\{M_p = 0, M_p = 1\} \times \{M_r = 1, M_r = 2\} \times \{M_a = 1, M_a = 2\}$$

which gives 8 LTSs for each case and each description style.

### 6.3 Model Generation Results

For verification purposes we need to know all the actions performed by the specified system. In this sense, the descriptions given for the second case can be used for verification, while the descriptions of the first case cannot be used because all visible actions are transformed into internal actions in order to model the independence from the environment. For this reason we generate only the models corresponding to the second case.

The results of LTSs generation are given in Table 1 for the process-oriented description, and in Table 2 for the data-oriented description. For each experiment, the tables give the size (in number of states and transitions) of the LTS and the time (in hours, minutes, and seconds) required for its generation.

All the experiments were performed on a Sun Ultra Sparc-1 machine (143 MHz) with 256 Mbytes of memory.

| $M_p$ | $M_r$ | $M_a$ | *LTS* | | *time* |
|---|---|---|---|---|---|
| | | | *states* | *trans.* | |
| 0 | 1 | 1 | 5,890 | 16,130 | $0:00'06''$ |
| 0 | 1 | 2 | 39,371 | 170,754 | $0:00'29''$ |
| 0 | 2 | 1 | 25,846 | 96,430 | $0:00'23''$ |
| 0 | 2 | 2 | 323,459 | 1,826,512 | $0:05'51''$ |
| 1 | 1 | 1 | 7,698,453 | 35,655,750 | $2:12'50''$ |
| 1 | 1 | 2 | $>$ 6,531,532 | $>$ 49,232,000 | $2:44'29''$ |
| 1 | 2 | 1 | $>$ 8,213,739 | $>$ 49,896,000 | $6:23'02''$ |
| 1 | 2 | 2 | $>$ 4,524,531 | $>$ 43,904,000 | $3:57'10''$ |

Table 1: Results of LTSs Generation for the Process-Oriented Style for Case 2

The experiments done for the process-oriented descriptions are limited by the state explosion of the model: the three last rows of table 1 give the sizes of the incomplete LTSs generated before the computing resources have been exhausted. The reason of the state explosion is the high degree of interleaving for this description: the only synchronization between the *Order* and the *Stock* processes is by means of

| $M_p$ | $M_r$ | $M_a$ | *LTS* | | *time* |
|---|---|---|---|---|---|
| | | | *states* | *trans.* | |
| 0 | 1 | 1 | 14,975 | 20,195 | $0:00'27''$ |
| 0 | 1 | 2 | 88,023 | 165,792 | $0:03'58''$ |
| 0 | 2 | 1 | 82,403 | 117,386 | $0:05'33''$ |
| 0 | 2 | 2 | 848,067 | 1,603,478 | $2:45'19''$ |
| 1 | 1 | 1 | > 5,236,886 | > 9,761,401 | $532:15'23''$ |

Table 2: Results of LTSs Generation for the Data-Oriented Style for Case 2

the *Withdraw* gate. For the same class of experiments, note that the LTS model increases with one order of magnitude when $M_r$ (maximal numbers of orders) or $M_a$ (maximal numbers of amounts) are incremented. Moreover, it increases more sharply with $M_a$ than with $M_r$.

The experiments done for the data-oriented descriptions are limited by the generation time: the last experiment (case $M_p$, $M_r$, and $M_a$ equal to 1) does not terminate after one week but does not explode. The reason is that the interleaving of the first description is transformed here in data computations. The functions used by the *Invoicing* process (see figure 3) perform several traversals over the lists containing the orders and the stocks, which are large structures. It is interesting to note that the LTS model increases with one order of magnitude when $M_r$ or $M_a$ are incremented. Contrary to the process-oriented style, the LTS model increases more sharply with $M_r$ than with $M_a$.

# 7 Verification using Temporal Logic

After generating the LTSs corresponding to the descriptions, we perform verification using temporal logic. More precisely, we use this approach to formalize the five informal statements given in the description of the case study and also to obtain other formal requirements.

As the dynamic semantics of LOTOS is action-based, it is natural to express the properties of the LOTOS descriptions in a temporal logic interpreted over the actions of LTSs. We used here a simplified fragment of the ACTL (Action CTL) temporal logic defined in [NV90], which is sufficiently powerful to express safety and liveness properties. The fragment of ACTL that we use has six primitive temporal operators (besides usual logic connectors):

$$\varphi \quad ::= \quad \mathbf{EX}_\alpha \varphi \;\mid\; \mathbf{AX}_\alpha \varphi \;\mid\; \mathbf{E}\,[\varphi_\alpha \mathbf{U} \varphi'] \;\mid\; \mathbf{A}\,[\varphi_\alpha \mathbf{U} \varphi'] \;\mid\; \mathbf{A}\,[\varphi_\alpha \mathbf{U}_{\alpha'} \varphi'] \;\mid\; \mathbf{E}\,[\varphi_\alpha \mathbf{U}_{\alpha'} \varphi']$$

where $\alpha$ is a set of transition labels. Informally, $\mathbf{EX}_\alpha \varphi$ (resp. $\mathbf{AX}_\alpha \varphi$) means "some (resp. every) path reaches $\varphi$ through an $\alpha$ step", $\mathbf{E}\,[\varphi_\alpha \mathbf{U} \varphi']$ (resp. $\mathbf{A}\,[\varphi_\alpha \mathbf{U} \varphi']$) means "some (resp. every) path stays in $\varphi$ through $\alpha$ steps until it reaches $\varphi'$", and $\mathbf{E}\,[\varphi_\alpha \mathbf{U}_{\alpha'} \varphi']$ (resp. $\mathbf{A}\,[\varphi_\alpha \mathbf{U}_{\alpha'} \varphi']$) means that "some (resp. every) path stays in $\varphi$ through $\alpha$ steps until it reaches $\varphi'$ through an $\alpha'$ step".

For commodity, we use the classical "diamond" and "box" modal operators, defined by $\langle\alpha\rangle\,\varphi = \mathbf{EX}_\alpha \varphi$ and $[\alpha]\,\varphi = \neg\mathbf{EX}_\alpha \neg\varphi$ respectively. We also use the derived operators $\mathbf{EF}_\alpha \varphi$ for $\mathbf{E}\,[\mathbf{true}_\alpha \mathbf{U}\ \varphi']$ and $\mathbf{AG}_\alpha \varphi$ for $\neg\mathbf{EF}_\alpha\ \neg\varphi$.

Also, we introduce the following shorthand notations:

- "$\mathbf{INEV}(\alpha_1, \alpha_2) = \mathbf{A}\,[\mathbf{true}_{\alpha_1} \mathbf{U}_{\alpha_2} \mathbf{true}]$", meaning that the program eventually performs an action satisfying $\alpha_2$, possibly preceded only by actions satisfying $\alpha_1$. Informally, "$\mathbf{INEV}(\alpha_1, \alpha_2)$" ensures the reachability of $\alpha_2$ independently from the (fair or unfair) scheduling policy of actions.

- "$\mathbf{FAIR}(\alpha_1, \alpha_2) = \mathbf{AG}_{\neg\alpha_2 \,\wedge\, \alpha_1}\ \mathbf{EF}_{\alpha_1}\ \langle\alpha_2\rangle\ \mathbf{true}$", meaning that every sequence of actions which do not satisfy $\alpha_2$, but satisfy $\alpha_1$, leads to a state from which it is possible to reach an action satisfying $\alpha_2$. Informally, "$\mathbf{FAIR}(\alpha_1, \alpha_2)$" means that, assuming a fair scheduling of actions, the program will eventually reach via actions satisfying $\alpha_1$ an action satisfying $\alpha_2$.

The last two operators are the action-based translations of the corresponding state-based operators defined in the LTAC temporal logic [QS83].

The rest of this section gives an overview of XTL [Mat98, MG98], describes the properties that we write using the ACTL library of XTL, and finally, discusses the verification results obtained.

## 7.1 Overview of XTL

Though primarily intended for evaluation of temporal logic formulas, XTL is in full generality a compiler for a functional language applied to a labelled transition system. The XTL language is equipped with data types for states, transitions and labels, and sets thereof, and functions for manipulating them (e.g., initial state, incoming and outgoing transitions of a state, source and target states of a transition). It can also deconstruct the labels of structured transitions and thus access the individual attributes of LOTOS events. Results are reported using a side-effect `print` function.

For example, the two following XTL expressions compute the set of all non deterministic states and search for some transition on *Deposit* with an amount larger than 10, respectively:

{S : **state where**
  **exists** T1 : **edge among out**(S), T2 : **edge among out**(S) **where**
    (T1 <> T2) **and** (**label**(T1)=**label**(T2))
  **end_exists**
}
**exists** T : **edge in**
    T → [DEPOSIT _ ?amt : **integer where** amt > 10]
**end_exists**

The notation "_" is used to represent a wild-card offer. Temporal operators are defined as functions and/or macros using these primitives; definitions for standard logics are provided as libraries of temporal operators.

## 7.2 The Correctness Properties

In the informal description there are no formal requirements against which the invoicing descriptions might be verified.

In order to obtain such properties, we consider the revised statements given in section 3.2. These statements can be split in three (non disjoint) classes:

- the first class contains statements which cannot be used to obtain formal requirements: R0.0, R0.1, R2.0, and R2.1.
- the second class contains statements which are obviously verified by the LOTOS descriptions: R0.2, the first part of R0.3, R0.4, R0.5, the first part of R2.2, and R2.3.
- the third class contains statements which lead to correctness properties: R0.3, R2.2, and R2.3.

We can also check classical properties (like deadlock free and liveness). In the following we present the properties obtained and the problems raised by their formalization or verification.

From the statement R0.3, we can formalize two properties. The first one is a safety property[14] and expresses that the quantity carried by an order should be a positive integer:

**Property 1.** *All the Request actions have strictly positive amounts.* The XTL formula below expresses this property.

**not** **exists** L : **label in**
    *L* → [REQUEST _ _ ?amt : **integer where** amt ≤ 0]
  **end_exists**

The second property extracted from the statement R0.3 is also a safety property and says that an order reference is unique. This is expressed by checking that duplicate references to an order (i.e., more than one action *Request* with the same parameter *ref*) appear if and only if the order has not been invoiced (i.e., a *Cancel* action for the order *ref* has been accepted):

**Property 2.** *Between two subsequent Request actions with the same reference ref* ($0 \leq ref \leq M_r - 1$), *a Cancel action with parameter ref must occur.* The XTL formula below (which uses ACTL operators) expresses this property.

**forall** ref : **integer among 0 ...** $M_r - 1$
  [REQUEST !ref ...] $\mathbf{AG}_{\neg(\text{CANCEL !ref})}$ [REQUEST !ref ...] **false**
**end_forall**

[14] Informally, safety properties specify that "something bad never happens".

The notation "..." in the label "REQUEST !ref ..." is used to represent a list of wild-card offers.

From the statement R2.2, we can formalize a safety property which states that only cancellations of existing orders are accepted:

**Property 3.** *A Cancel action with the parameter ref* $(0 \leq ref \leq M_r - 1)$ *can appear only if a Request action of reference ref has been done.*

$$
\begin{array}{l}
\textbf{let } \varphi : \textbf{stateset} = \ \textbf{forall} \text{ ref} : \textbf{integer among 0 ...} \ M_r - 1 \\
\qquad\qquad\qquad\qquad \textbf{AG}_{\neg(\text{REQUEST !ref ...})} \ [\text{CANCEL !ref}] \ \textbf{false} \\
\qquad\qquad\qquad\quad \textbf{end\_forall} \\
\textbf{in} \\
\quad (\textbf{init} \Longrightarrow \varphi) \ \wedge \ \ [\text{CANCEL ...}] \ \varphi \\
\textbf{end\_let}
\end{array}
$$

The "**init**" operator of XTL characterizes the initial state of an LTS. It is useful in order to express more naturally certain properties (e.g., past-tense properties).

From the statement R2.3, we can formalize a liveness property[15] stating that an order which is not cancelled will be eventually invoiced. We remark that this property expresses requirements on the state variables of the LOTOS description: in the process-oriented description, this is equivalent to testing the *sta* parameter of the *Order* process, in the data-oriented description this is equivalent to testing the *ords* parameter of the *Invoice* process. Because the underlying model of LOTOS is the LTS model (i.e., does not contain information in the states), we cannot directly test the state variables, but we can access them only by means of transitions. For this reason, we add in the initial descriptions some transitions signaling the change of the status of an order. We present below only the modification of the process-oriented description, which is simpler; the modification of the data oriented description needs more complex changes. The modification consists in introducing an *Invoice* action in the process *Order* before the recursive call of the process which changes the *sta* parameter in *Invoiced*:

```
process Order [Request, Cancel, Invoice, Withdraw]
 (ref:Reference, prd:Product, amt:Amount, sta:Status) : noexit :=
  [sta = None] ->
    Request !ref ?prd:Product ?amt:Amount [amt gt 0];
    Order [Request, Cancel, Withdraw] (ref, prd, amt, Pending)
[]
  [sta = Pending] ->
    (
      Cancel !ref;
      Order [Request, Cancel, Withdraw] (ref, 0 of Product, 0 of Amount, None)
    []
      Withdraw !prd !amt;
      Invoice !ref; (* added transition *)
      Order [Request, Cancel, Withdraw] (ref, prd, amt, Invoiced)
    )
endproc
```

Then, the property can be written as follows:

**Property 4.** *After a Request action with the reference ref* $(0 \leq ref \leq M_r - 1)$*, if a Cancel action with the same parameter ref does not execute, then eventually an Invoice action with parameter ref will be reached.*

In order to ignore unfair executions (e.g., those in which an order is always cancelled before invoicing), we must use "**FAIR**" instead of "**INEV**" to express the reachability of the *Invoice* action.

$$
\begin{array}{l}
\textbf{init} \ \Longrightarrow \ \textbf{forall} \text{ ref} : \textbf{integer among 0 ...} \ M_r - 1 \\
\qquad\qquad [\text{REQUEST !ref ...}] \, \textbf{FAIR}(\neg(\text{CANCEL !ref}), \text{INVOICE !ref}) \\
\qquad\quad \textbf{end\_forall}
\end{array}
$$

In the informal requirements nothing is said about the safety and the liveness of the general system. For this reason we add the following two properties. The first one is a liveness property and checks if

[15] Informally, liveness properties specify that "something good eventually happens".

the execution can advance infinitely, i.e., there are no deadlocks.
**Property 5.** *The system is deadlock free.*

$$\neg\,[\mathbf{true}]\,\mathbf{false}$$

The second property is imposed by the existence of the internal actions (noted "**i**") in the LOTOS descriptions. The internal actions appear in the LTS models of the process-oriented description from the hiding of the *Withdraw* gate. In the data oriented description, internal actions are generated by the enabling operator ">>". The safety property below checks that there are no livelocks (cycles of internal actions), i.e., every path consisting of internal actions eventually reaches a visible action.
**Property 6.** *The description does not diverge, i.e., there are no cycles of internal actions.*

$$\mathbf{AG_i A}\left[\mathbf{true_i U_{\neg i} true}\right]$$

### 7.3 Verification Results

The six formulas given in section 7.2 are evaluated on the LTSs models completely generated (before and after the introduction of the *Invoice* action) using the XTL [Mat98, MG98] prototype model-checker.

It is worth noticing that, since the XTL language allows the definition of macro-notations (for action predicates as well as for temporal operators), the ACTL formulas given in section 7.2 are almost identical to those written in the XTL source code.

The verifications were performed for all the LTSs produced by model generation (see section 6) and minimized modulo strong equivalence. For each LTS, the time needed for evaluating the five formulas was less than one minute.

The properties 1–3 and 5–6 are true on all LTSs described in section 6. The six properties are also true on the LTSs obtained after the introduction of the *Invoice* action.

### 7.4 Questions on Requirements

In conclusion to the formalisation of the verification properties, we may raise three new questions about the informal requirements:

**Q21.** Is an uncancelled order eventually satisfied? This is not stated in the informal requirements given in section 1 or in the revised informal requirements given in section 3.1. The change of the state of an invoiced order may be infinitely delayed by the system. In our case, this is possible if the system is not fair, which is not a reasonable assumption. When the system is fair, we verify on a slightly modified version of the LOTOS descriptions that an order is eventually satisfied.

**Q22.** May the system deadlock? This is presumably not a desired property of the system. In our case, we choose the negative answer and we verify it on the LTS models generated.

**Q23.** May the system diverge in Case 2? That is, may the system do only internal actions without accepting any interaction with the environment? For the Case 1, the answer is given in the requirement R1.2. For the Case 2, we choose the negative answer and verify it on the LTS models.

## 8 Verification using Bisimulations

As an alternative to verification using temporal logics, we also performed verification using bisimulations by means of the ALDÉBARAN tool.

Our goal was to prove that process-oriented and data-oriented LOTOS descriptions where "equivalent" by checking the equivalence between the corresponding LTSs.

The ALDÉBARAN tool provides a lot of bisimulation equivalences: strong equivalence [Par81] (noted $\sim$), branching equivalence [vGW89] (noted $\approx_b$), delay equivalence [vGW89] (noted $\approx_d$), observational equivalence [Mil89] (noted $\approx_o$), $\tau^*a$ equivalence [FM90] (noted $\approx_{\tau^*a}$), and safety equivalence [Rod88, BFG+91] (noted $\approx_s$). Figure 4 gives the lattice (the upper relation being finer) of these equivalence relations. We tried to check the equivalence of the two corresponding LTSs modulo all the equivalences offered by ALDÉBARAN. Since the LTSs of the two descriptions contain internal actions, the

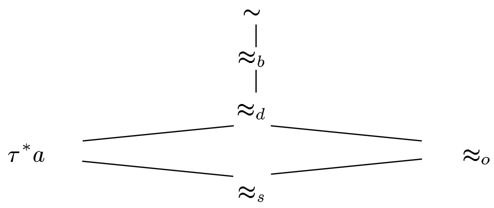

Figure 4: The lattice of the equivalences supported by the ALDÉBARAN tool

branching and the observational equivalences cannot be used because they need that at least one of two LTS does not contain internal actions.

We found that the LTS of the data-oriented version is included in *but not equivalent to* the corresponding LTS of the process-oriented style modulo the safety relation. The equivalence is broken because in the process-oriented descriptions a *Cancel* action can be done after a *Deposit* action if the order is not yet treated, i.e., the following sequence is a valid one:

```
i*
REQUEST !0 !0 !1
i*
DEPOSIT !0 !1
CANCEL !0
```

In the data-oriented specification this sequence of actions is not possible because the list of orders is updated after each *Deposit* action at the recursive call of the *Invoice* process (see section 4.4).

In order to obtain the same behaviour for the data-oriented description, we modify the process *Invoice* as follows: (1) by eliminating the update of stocks and orders at its recursive call and (2) by introducing an explicit, internal action (the fourth case of the choice below) which explicitly updates the lists of stocks and orders.

```
process Invoice [Request, Cancel, Deposit] (ords:Orders, stks:Stocks) : noexit :=
 (
    Request ?ref:Reference ?prd:Product ?amt:Amount [(StatOrder(ref, ords) eq None) and (amt gt 0)];
    exit (AddOrder(MkOrder(ref, prd, amt, Pending), ords), stks)
  []
    Cancel ?ref:Reference [StatOrder(ref, ords) eq Pending];
    exit (RemOrder(MkOrder(ref, 0, 0, Pending), ords), stks)
  []
    Deposit ?prd:Product ?amt:Amount [amt gt 0];
    exit (ords, AddStock(MkStock(prd, amt), stks))
  []
    exit (UpdateOrders(ords, stks), UpdateStocks(ords, stks))
  )
>>
  accept newords:Orders, newstks:Stocks in
    Invoice [Request, Cancel, Deposit] (newords, newstks)
endproc
```

This modification introduces livelocks in the LTSs of the data-oriented description. These livelocks appear when the updating is chosen although neither the set of orders, nor the set of stocks are modified (by a *Request*, a *Cancel*, or a *Deposit* action). In this case, the internal action generated by the enable operator (>>) loops in the same state. That means that the sixth property given in section 7.2 is invalidated on these LTSs. However, this allows us to obtain a data-oriented description which is safety equivalent to the process-oriented one. This means that the two descriptions are equivalent for all safety properties, and all actions executed by one description are correct with respect to the other description.

# 9 Conclusion

In this report, we demonstrate how LOTOS, E-LOTOS, and the CADP toolbox may be used for requirement analysis, formal description and verification, by applying them to the Invoicing case study.

This experience conduct us to formulate twenty-three questions on the informal description of the Invoicing system [AAH98]. These questions arise either from the formal description of the system, or from the verification of the description.

*Concerning formalisation*, the LOTOS philosophy based on the "black box" principle is very useful to obtain a high-level specification of requirements. For the Invoicing case study, twenty questions arise when trying to formalise the informal requirements.

Also, LOTOS shares its behavioural approach with process algebras such as CCS [Mil89] and CSP [Hoa85]. There are thus a number of languages that might be used in the same kind of style. However, LOTOS is relatively unusual in having an integration of behaviour with data specification (ACTONE [EM85]). This is convenient for specification since different aspects of a problem can be treated as process or data. The process-oriented specifications in this report show that this can be an effective mix.

Of the four Case 2 descriptions, the authors are most satisfied with the E-LOTOS process-oriented description. It is clear that E-LOTOS offers a much cleaner and more compact style of specification compared to current LOTOS. In particular, modularity, typed gates, and functional data types are felt to be much preferable. The data types used in LOTOS (based on ACTONE) have been rather disliked for the verbosity that is evident in the specifications of this report. The LOTOS data type library is also somewhat distant from conventional programming practice. Some syntactic LOTOS data typing shorthands have been developed for these reasons [Pec93].

The process-oriented and data-oriented specifications make an interesting comparison. In E-LOTOS there is little to choose between them regarding clarity or compactness. However in LOTOS, the data-oriented specification is tedious to read because of the verbose data part. In general, there are good reasons to prefer the process-oriented approach. It takes an object-based view, and thus is closer to current analysis practice. The approach also hints at possible concurrent or distributed implementation, and thus may be closer to engineering practice.

*Concerning verification*, the CADP toolbox offers for LOTOS the state-of-the-art verification techniques: to the best of our knowledge, the CÆSAR [GS90] and CÆSAR.ADT [Gar89] compilers are the only model checkers supporting dynamic data structures (such as lists). Two main forms of verification are supported: bisimulations and temporal logics. Thus, for the Invoicing case study we generate the Labelled Transition Systems (LTSs) of the LOTOS descriptions using the CÆSAR and CÆSAR.ADT compilers. Then, we perform verification by the two approaches of CADP:

- temporal logics, using the ACTL [NV90] library of the XTL [Mat98, MG98] model-checker. For this, we establish six formal properties: absence of deadlocks and livelocks, and four logical ordering properties between visible actions. These properties have been extracted from three of the revised informal statements resulted from formalisation, expressed in XTL (using the ACTL operators), and then checked on the generated models.
- bisimulations, using the ALDÉBARAN [Mou92] tool. Firstly, we use strong bisimulation [Par81] to reduce the LTSs generated with CÆSAR. Secondly, we show that the LTSs corresponding to the initial description cannot be equivalent modulo the equivalence relations offered by ALDÉBARAN. However, by performing a slight modification of the data-oriented description, we obtain the safety equivalence of the two descriptions.

Using model-checking verification, we have to cope with its main limitation, the state explosion problem. In order to generate finite LTSs of tractable size, we have to modify the initial descriptions in the following points:

- We transform the ACTONE types to obtain constructive definitions for sorts. This is not a real problem since the new version of LOTOS, E-LOTOS, supports constructive (ML-like) type declarations.
- We limit the unbounded parallelism to a finite number of parallel processes.

- We restrict the user-defined data domains to finite domains.

These restrictions do not suffice to avoid state explosion for the model. The reason for this explosion is the high degree of non-determinism in the process-oriented descriptions and the large data structures in the data-oriented descriptions. We hope that the ongoing research in the group of CADP toolbox will allow us to avoid this state explosion by producing reduced LTSs.

We also have to modify the initial descriptions in order to be able to verify properties based on the state of variables or to obtain (safety) equivalent descriptions.

It would not be wise to claim that any formal method was "better" than any other. However, it is hoped that this report has convinced about the suitability of our formal methods and tools for the requirement analysis, formal description, and verification.

### Acknowledgments

Thanks are due to Carron Shankland (University of Stirling), Hubert Garavel (INRIA), and Radu Mateescu (INRIA) for their suggestions and carefully reviewing of a draft version of this report. Henri Habrias (University of Nantes) also kindly offered advice on a draft of the papers forming the basis of this report.

## References

[AAH98] M. Allemand, C. Attiogbé, and H. Habrias, editors. *International Workshop on Comparing System Specification Techniques.* University of Nantes (France) and IRIN, IRIN, March 1998.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

[BFG+91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. Safety for Branching Time Semantics. In *Proceedings of 18th ICALP*. Springer Verlag, July 1991.

[BLV95] Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris A. Vissers, editors. *The LOTOSphere Project.* Kluwer Academic Publishers, London, UK, 1995.

[CGM+96] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, October 1996. Full version available as INRIA Research Report RR-2958.

[Cla92] R. G. Clark. LOTOS Design-Oriented Specification in the Object-Based Style. Technical Report CSM-84, Department of Computing Science and Mathematics, University of Stirling, FK9 4LA Stirling, Scotland, April 1992.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to Algebraic Specifications Based on the Language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[Eij89] Peter H. J. van Eijk. Tools for LOTOS Specification Style Transformation. In S. T. Vuong, editor, *Proc. Formal Description Techniques II.* North-Holland, Amsterdam, Netherlands, December 1989.

[Eij91] P. H. J. van Eijk. The LotoSphere Integrated Tool Environment Lite. In K. R. Parker and G. A. Rose, editors, *Proceedings of the 4th International Conference on Formal Description Techniques FORTE'91 (Sydney, Australia)*, pages 471–474. North-Holland, November 1991.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[FKM93] Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Symbolic Equivalence Checking. In C. Courcoubetis, editor, *Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece)*, volume 697 of *Lecture Notes in Computer Science*. Springer Verlag, June 1993.

[FLS97] Mohammed Faci, Luigi Logrippo, and Bernard Stepien. Structural Models for specifying Telephone Systems. *Computer Networks and ISDN Systems*, 29(4):501–528, March 1997.

[FM90] Jean-Claude Fernandez and Laurent Mounier. Verifying Bisimulations "On the Fly". In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*. North-Holland, November 1990.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[GJM+97] Hubert Garavel, Mark Jorgensen, Radu Mateescu, Charles Pecheur, Mihaela Sighireanu, and Bruno Vivien. CADP'97 – Status, Applications and Perspectives. In Ignac Lovrek, editor, *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, June 1997.

[GR90] Hubert Garavel and Carlos Rodríguez. An Example of LOTOS Specification: The Matrix Switch Problem. Rapport SPECTRE C22, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, June 1990.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[GS98] Hubert Garavel and Mihaela Sighireanu. Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS. In Jan-Friso Groote, Bas Luttik, and Jos van Wamel, editors, *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems FMICS'98 (Amsterdam, The Netherlands)*, pages 187–230, Amsterdam, May 1998. CWI. Invited lecture.

[Gut77] J. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, 20(6):396–404, June 1977.

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

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

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[ISO92] ISO/IEC. Distributed Transaction Processing — Part 3: Protocol Specification. International Standard 10026-3, International Organization for Standardization — Information Technology — Open Systems Interconnection, Genève, 1992.

[Led87] Guy J. Leduc. The Intertwining of Data Types and Processes in LOTOS. In Harry Rudin and Colin H. West, editors, *Proc. Protocol Specification, Testing and Verification VII*, pages 123–136. North-Holland, Amsterdam, Netherlands, May 1987.

[LL95] R. Lai and A. Lo. An Analysis of the ISO FTAM Basic File Protocol Specified in LOTOS. *Australian Computer Journal*, 27(1):1–7, February 1995.

[Mat98] Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, April 1998.

[MG98] Radu Mateescu and Hubert Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In Tiziana Margaria, editor, *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98 (Aalborg, Denmark)*, July 1998.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mou92] Laurent Mounier. *Méthodes de vérification de spécifications comportementales : étude et mise en œuvre*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1992.

[Mun91] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[NV90] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[Pec93] Charles Pecheur. Vlib: Infinite Virtual Libraries for LOTOS. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 29–44. North-Holland, Amsterdam, Netherlands, May 1993.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[Que90] Juan Quemada. The Two-Key System: Playing with Styles in LOTOS. Technical Report, Department of Telematic Engineering, Polytechnic University of Madrid, Spain, March 1990.

[Que98] Juan Quemada, editor. Committee Draft on Enhancements to LOTOS. ISO/IEC FCD 15437 (E-LOTOS) (also SC33 N188), April 1998.

[Rod88] Carlos Rodríguez. *Spécification et validation de systèmes en XESAR.* Thèse de Doctorat, Institut National Polytechnique de Grenoble, May 1988.

[Sco93] Giuseppe Scollo. *On the Engineering of Logics.* PhD thesis, University of Twente, Enschede, Netherlands, March 1993.

[Sin89] Marten van Sinderen. A Verification Exercise Relating to Specification Styles in LOTOS. Technical Report INF-89-18, University of Twente, Enschede, Netherlands, March 1989.

[SL93] Bernard Stepien and Luigi Logrippo. Status-Oriented Telephone Service Specification: An Exercise in LOTOS Style. In Teodor Rus and Charles Rattray, editors, *Theories and Experiences for Real-Time System Development*, Computing: Vol 2, pages 1–21. World Scientific, October 1993.

[TS95] Kenneth J. Turner and Marten van Sinderen. LOTOS Specification Style for OSI. In Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris A. Vissers, editors, *The LOTOSphere Project*, pages 137–159. Kluwer Academic Publishers, London, UK, 1995.

[Tur90] Kenneth J. Turner. A LOTOS-Based Development Strategy. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 157–174. North-Holland, Amsterdam, Netherlands, 1990.

[Tur93a] Kenneth J. Turner. An engineering approach to formal methods. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 357–380, Amsterdam, Netherlands, June 1993. North-Holland.

[Tur93b] Kenneth J. Turner, editor. *Using Formal Description Techniques – An Introduction to ESTELLE, LOTOS, and SDL.* John Wiley, 1993.

[Tur97a] Kenneth J. Turner. Incremental Requirements Specification with LOTOS. *Requirements Engineering Journal*, 2:132–151, November 1997.

[Tur97b] Kenneth J. Turner. Relating Architecture and Specification. *Computer Networks and ISDN Systems*, 29(4):437–456, March 1997.

[Tur97c] Kenneth J. Turner. Specification architecture illustrated in a communications context. *Computer Networks and ISDN Systems*, 4(29):397–411, March 1997.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[Viv97] Bruno Vivien. Etude et réalisation d'un compilateur E-LOTOS à l'aide du générateur de compilateurs SYNTAX/FNC-2. Mémoire d'ingénieur, CNAM, Grenoble, December 1997.

[VSS91] Chris A. Vissers, Giuseppe Scollo, and Marten van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. *Theoretical Computer Science*, 89:179–206, 1991.

[VSSB90] Chris A. Vissers, Giuseppe Scollo, Marten van Sinderen, and Ed Brinksma. On the Use of Specification Styles in the Design of Distributed Systems. Technical Report, Department of Informatics, University of Twente, Enschede, Netherlands, 1990.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399