Kenneth J. Turner. The Invoicing Case Study in (E-)LOTOS. In Michel Allemand, Christian Attiogbe and Henri Habrias, editors, Proc. International Workshop on Comparing Systems Specification Techniques, pages 83-98, University of Nantes, France, March 1998.

The Invoicing Case Study in (E-)LOTOS

Kenneth J. Turner

Department of Computing Science and Mathematics
University of Stirling, Stirling FK9 4LA, Scotland

Telephone: +44-1786-467-420 *Facsimile:* +44-1786-464-551
Email: kjt@cs.stir.ac.uk *Web:* http://www.cs.stir.ac.uk/~kjt/

1st June 1999

Abstract

The informal requirements for the invoicing case study are analysed and interpreted. This leads to a high-level specification architecture that can be formalised. Specifications are presented in LOTOS (Language Of Temporal Ordering Specification). For comparison, specifications are also presented E-LOTOS (Enhancements to LOTOS) – the new version of LOTOS currently being standardised. Since LOTOS allows a balance to be struck between process-oriented and data-oriented modelling, specifications in both styles are given. The resulting specifications are evaluated in the context of LOTOS and formal approaches more generally.

Keywords: Invoicing, E-LOTOS (Enhancements to LOTOS), Formal Method, LOTOS (Language Of Temporal Ordering Specification)

1 Introduction

This paper presents an analysis of the invoicing case study proposed by Henri Habrias (University of Nantes). The purpose of this case study is to see what questions are raised by the application of formal methods to a small but possibly realistic example of informal requirements. The paper gives specifications in LOTOS (Language Of Temporal Ordering Specification [5]) and E-LOTOS (Enhancements to LOTOS [6]). A companion paper by Mihaela Sighireanu [11] investigates properties of these specifications formally. Informal requirements were given as follows, as numbered by the author for later reference (R*n*).

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.

R1. Case 1:
- R1.1 All the ordered references are in stock.
- 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 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.

2 The LOTOS Approach

2.1 The LOTOS Philosophy

The LOTOS Position^{TM} is to treat a system as 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). Methods have been investigated for LOTOS, e.g. [1, 14], but because the case study was so small the author 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.

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 [2, 4, 9, 13, 15, 16, 17, 19, 20, 21, 22, 23]. 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[1] of a system. Several LOTOS workers have considered general 'quality' principles for specification architecture [10, 18].

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 [7]. 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 requirements 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-functionality: What timing and performance aspects must be specified? What other organisational issues should be considered?

Specification: How should the formal model be developed? What specification style is appropriate? How should the specification be validated or verified?

The case study deals with requirements capture, analysis and specification of the invoicing system. 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 author to raise questions about the invoicing requirements and to answer them himself in a sensible fashion.

[1] In the sense of [18], the architecture of a specification means its structure and style.

2.2 Requirements Questions

The following questions arose when trying to formalise the requirements for the invoicing case study. The questions (Q*n*) are presented according to the classification in section 2.1, though they did not arise in a strict hierarchical order. Requirements (R*n*) are taken from the informal statements in section 1. As will be seen, the volume of questions is much greater than the informal problem statement!

Environment

Q1. How many users are there of the order-invoice 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 (or other things for users), it would be necessary to identify users or orders. Invoices and the like 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, being 'given stock and the set of orders' (R1.3) means that the first case study deals with a system that does not directly accept stock or order 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 stocks 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 stocks 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 stocks 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 cite 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 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.[2]

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

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

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. This means that when the system is given 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 stock should be allocated. Again, an '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 system is 'given stock and orders 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 stocks 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 they may be permitted by the specification.

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-functionality

Q19. Client requirements would normally include non-functional aspects such as cost, delivery schedule, performance, reliability, integration and testing. Performance specification and testing have been studied in LOTOS-based development. However, non-functional aspects can be ignored since the only requirements available are strictly functional.

Specification

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 requirements specifications are desired. However, it seems sensible to treat the first case as a less detailed form of the second.

2.3 Revised Problem Statement

The informal problem statement in section 1 was clarified using the answers in section 2.2. 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 of the invoicing of orders. You are not responsible for actually issuing invoices or delivering products.
- R0.2 To invoice is to change the state of an order (to change its explicit status 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 your sub-system and the user.
- R0.4 The same product reference can appear in several different orders, some of which 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:

- R2.0 You should treat this as the primary case, of which Case 1 is a simplification.
- R2.1 You do have to take into account the entry of new orders, cancellations of orders, and entries of quantities in the stock.
- R2.2 A cancellation cites 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.

2.4 The (E-)LOTOS Specifications

The case study is mainly data-oriented since it effectively describes a database. For this reason, its 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.

A new version of LOTOS is currently being standardised by ISO as E-LOTOS. Among many improvements on current LOTOS, E-LOTOS introduces modules, typed gates and better data typing. Since E-LOTOS is still undergoing standardisation, some of its constructs are still to be stabilised. The author has assumed that a fully imperative semantics will be introduced (following the proposal of researchers at INRIA Rhône-Alpes). Among other things, this simplifies the specification of loops. The author has also assumed the existence of an array type which is not yet in E-LOTOS but is a likely addition.

Since E-LOTOS is the future form of the language, the author felt it would be interesting to see how its specification style differed from LOTOS. E-LOTOS specifications are presented first in sections 3 and 4, followed

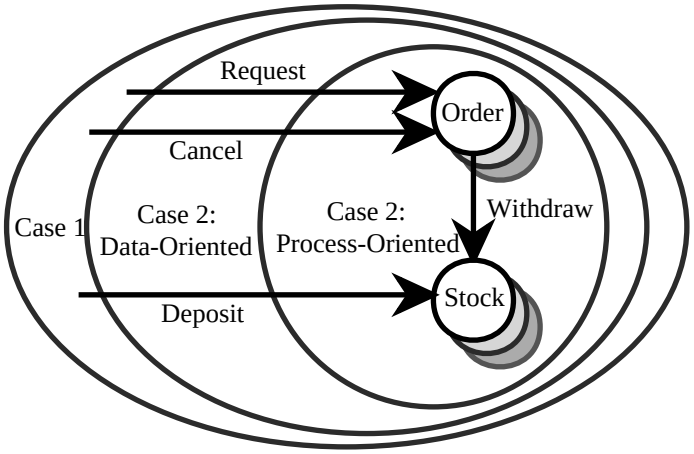

Figure 1: Structure of Specifications

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 author has used his own convention for the case of identifiers (variables in lower case, other identifiers with an initial capital).

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 approach so the load on the reader should not be too high.

As argued in section 2.2, 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).

3 Case 2

Since Case 1 is treated as just an abstraction of Case 2, the full case study is specified first in this section then an abstract form is given in section 4.

3.1 Process-Oriented E-LOTOS Specification

LOTOS models a system as a collection of communicating processes, with data values and operations described by types. The communication ports of a process are called gates that, in E-LOTOS, are typed. Gate typing allows static checking of the kinds of values that are communicated. LOTOS processes are parameterised by their gates and state variables. Processes make event offers such as *Withdraw(!product,?amount)* 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 '?'. These notations are used more widely in the language for pattern-matching of expressions.

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.

The data types and processes are specified here in a separate module for convenience. 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 endtype
  type Product renames Nat endtype
  type Amount renames Nat endtype
```

The status of an order is defined using an enumerated type; *None* is used for an order that is not current. The complete type for an order can be then given as a record containing product, amount and status fields. A collection of orders is treated as an associative array indexed by order reference. A collection of stocks is similar, but the array is indexed by product code and the values are amounts.

```
  type Status is enum None, Pending, Invoiced endtype
  type Order is record Prod:Product, Amt:Amount, Stat:Status endtype
  type Orders is array of Order[Reference] endtype
  type Stocks is array of Amount[Product] endtype
```

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 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, Cancel, Withdraw]
   (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
    endloop
  endproc
```

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] (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
      endloop
    endvar
  endproc
```

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, Cancel, Withdraw] (ref:Reference) is
  Order [Request, Cancel, Withdraw] (ref, 0, 0, None)
|||
  Orders [Request, Cancel, Withdraw] (Succ(ref))
endproc
process Stocks [Deposit, Withdraw] (prd:Product) is
  Stock [Deposit, Withdraw] (prd, 0)
|||
  Stocks [Deposit, Withdraw] (Succ(prd))
endproc
endmod
```

The overall system 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 imports OrderStock is
  gates Request:(Reference, Product, Amount), Cancel:Reference, Deposit:(Product, Amount)
  behaviour
```

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.

```
    hide Withdraw:(Product, Amount) in
      Orders [Request, Cancel, Withdraw] (0)
    |[Withdraw]|
      Stocks [Deposit, Withdraw] (0)
endspec
```

3.2 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* $\Rightarrow$ *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 is similar to the 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
```

3.3 Data-Oriented E-LOTOS Specification

In this approach, orders and stocks are defined by data values rather than processes. Invoicing then becomes an operation on these values. A data type module is used, similar to that for the process-oriented version.

```
module OrderStock is
  type Reference renames Nat endtype
  type Product renames Nat endtype
  type Amount renames Nat endtype
  type Status is enum None, Pending, Invoiced endtype
  type Order is record Prod : Product, Amt : Amount, Stat : Status endtype
  type Orders is array of Order[Reference] endtype
  type Stocks is array of Amount[Product] endtype
```

Invoicing orders is carried out by a function that takes current orders and stocks. Each order is checked in a loop, from first reference number to last. The *Next* function finds the next array index since there may be gaps in order numbers. Orders are thus fulfilled in reference number sequence, and not non-deterministically as in the process-oriented version. Non-determinism could have been achieved, but by complicating the specification. The current reference is used to extract the product, amount and status of a record. The product code 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 the updated orders and stocks. If an order cannot be fulfilled, it may be satisfied later when invoicing is repeated on receipt of new stock.

```
  function Invoice(ords:Orders, stks:Stocks) : (Orders, Stocks) is
    var ref:Reference, prd:Product, amt,stk:Amount, sta:Status in
      for (?ref := First(ords); ref <= Last(ords); ?ref := Next(ords, ref)) do
        (?prd, ?amt, ?sta) := Get(ords, ref);
        ?stk := Get(stks, prd);
        if (sta == Pending) and (amt <= stk) then
          ?ords := Put(ords, ref, Set_Stat(ord, Invoiced));
          ?stks := Put(stks, prd, stk − amt)
        endif
      endfor
      (ords, stks)
    endvar
  endfunc
endmod
```

The system specification is like that for the process-oriented version except that local variables are introduced. In particular, orders and stocks are initialised as empty.

```
specification Invoicing imports OrderStock is
  gates Request:(Reference, Product, Amount), Cancel:Reference, Deposit:(Product, Amount)
  behaviour
    var ords:Orders := Empty, stks:Stocks := Empty, ref:Reference, prd:Product, amt:Amount in
```

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* array. Each branch of the loop updates orders or stocks as appropriate. The *Invoice* function is then called to schedule orders and alter stocks.

```
      loop
        (
          Request(?ref, ?prd, ?amt) [NotIn(ords, ref) and (amt > 0)];
          ?ords := Put(ords, ref, Order(prd, amt, Pending))
        []
          Cancel(?ref) [IsIn(ords, ref) andthen (Get_Stat(Get(ords, ref)) == Pending)];
          ?ords := Delete(ords, ref)
        []
          Deposit(?prd, ?amt) [amt > 0];
          ?stks := Put(stks, prd,
            if IsIn(stks, prd) then Get(stks, prd) + amt else amt endif)
        );
        (?ords, ?stks) := Invoice(ords,stks)
      endloop
    endvar
endspec
```

3.4 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*(inal) 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.[3] *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

[3] 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.

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* $\Rightarrow$ *equation*) apply only if the condition holds.

```
type Orders is Order, Status
  sorts Orders
  opns
    NoOrders : -> Orders
    AddOrder : Order, Orders -> Orders
    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));
        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 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 permits 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 newords:Orders, newstks:Stocks in
      Invoice [Request, Cancel, Deposit]
       (UpdateOrders(newords, newstks), UpdateStocks(newords, newstks))
  endproc
endspec
```

4 Case 1

Following the discussion in section 2.2, the first case is viewed as an abstraction of the second. 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.

4.1 Process-Oriented E-LOTOS Specification

The external gates of Case 2 are declared and hidden along with the internal gate.

```
behaviour
  hide Request:(Reference, Product, Amount), Cancel:Reference, Deposit:(Product, Amount),
   Withdraw:(Product, Amount) in
```

4.2 Process-Oriented LOTOS Specification

The external gates of Case 2 are removed from the specification heading and hidden along with the internal gate.

```
specification Invoicing : noexit

behaviour
  hide Request, Cancel, Deposit, Withdraw in
```

4.3 Data-Oriented E-LOTOS Specification

The external gates of Case 2 are declared and hidden.

```
hide Request:(Reference, Product, Amount), Cancel:Reference, Deposit:(Product, Amount) in
```

4.4 Data-Oriented LOTOS Specification

The external gates of Case 2 are removed from the specification heading and hidden.

```
specification Invoicing : noexit

behaviour
  hide Request, Cancel, Deposit in
```

5 Validation and Verification

Since E-LOTOS is currently being standardised, tools for the language are still under development and could not be used. The E-LOTOS specifications should hence be regarded as conceptual at this stage. However they are similar to the LOTOS specifications and have been independently reviewed, so there is a degree of confidence in them.

The LOTOS specifications have been validated using standard tools (LITE, CADP) 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 current stock, amounts exactly equal to 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. Validation 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.

There were no formal requirements against which the specifications might have been verified. Verification might have been undertaken in the sense that the state space of the specifications might have been analysed. Equivalence might also have been checked between the various specifications. It is claimed that the four Case 2 specifications are testing equivalent to each other. Formal properties of the specifications are not, however, analysed here. A companion paper by Mihaela Sighireanu [11] investigates the specifications using model-checking.

6 Discussion

Of the four Case 2 approaches, the author is most satisfied with the E-LOTOS process-oriented specification. 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 ACT ONE [3]) have been rather disliked for the verbosity that is evident in the specifications of this paper. 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 [8].

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.

LOTOS shares its behavioural approach with process algebras such as CSP (Communicating Sequential Processes) and CCS (Calculus of Communicating Systems). 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 (ACT ONE in LOTOS, ML in E-LOTOS). This is convenient for specification since different aspects of a problem can be treated as process or data. The process-oriented specifications in this paper show that this can be an effective mix.

Compared to model-based languages like B, VDM and Z, LOTOS offers concurrency and an operational view. LOTOS lacks the convenience of standard mathematical models such as sets and relations. However, the E-LOTOS library is being extended in this direction. The author has experience of using both LOTOS and Z to describe the same problem domain (the reference model for Open Distributed Processing [12]). As might be expected, both languages have something to offer. To over-simplify, LOTOS is more appropriate for specifying dynamic, processing-oriented aspects of a system whereas Z is more appropriate for static, data-oriented aspects.

It would not be wise to claim that any specification language was ‘better’ than any other. However, it is hoped that the paper has shown how LOTOS raises interesting questions and offers benefits for the invoicing case study.

Acknowledgements

Thanks are due to Carron Shankland (University of Stirling) for carefully reviewing a draft of the paper. The author is also very grateful to Mihaela Sighireanu (INRIA Rhône-Alpes) for the effort she put into checking the (E-)LOTOS specifications. Henri Habrias (University of Nantes) also kindly offered advice on a draft of the paper.

References

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

[2] Robert G. Clark. Using LOTOS in the object-based development of embedded systems. In Charles M. I. Rattray and Robert G. Clark, editors, *The Unified Computation Laboratory*, pages 307–319. Oxford University Press, May 1992.

[3] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1985.

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

[5] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.

[6] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Enhancements to LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC CD. International Organization for Standardization, Geneva, Switzerland, November 1997.

[7] 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.

[8] 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.

[9] Juan Quemada. The two-key system: Playing with styles in LOTOS. Technical report, Department of Telematic Engineering, Polytechnic University of Madrid, Spain, March 1990.

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

[11] Mihaela Sighireanu. Model-checking validation of the LOTOS descriptions of the invoicing case study. In Henri Habrias, editor, *Proc. International Workshop on Specification Techniques and Formal Methods*, Nantes, France, March 1998. University of Nantes, France.

[12] Richard O. Sinnott. An architecture based approach to specifying distributed systems in LOTOS and Z. Technical Report CSM-144, Department of Computing Science and Mathematics, University of Stirling, UK, September 1997.

[13] 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.

[14] 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.

[15] 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. North-Holland, Amsterdam, Netherlands, June 1993.

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

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

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

[19] 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.

[20] 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.

[21] Marten van Sinderen. A verification exercise relating to specification styles in LOTOS. Technical Report INF-89-18, University of Twente, Enschede, Netherlands, March 1989.

[22] 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.

[23] 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.