Contents

2 (E-)LOTOS: (Enhanced) Language Of Temporal Ordering Specification

Kenneth J. Turner and Mihaela Sighireanu

2.1 Overview of the LOTOS Notation and Method

This section introduces the LOTOS and E-LOTOS languages, and how they may be used in requirements capture.

2.1.1 The LOTOS and E-LOTOS Languages

LOTOS (Language of Temporal Ordering Specification [7]) is a standardised FDT (Formal Description Technique) originally intended for the specification of communications and distributed systems. Several tutorials for LOTOS are available [1,16]. The design of LOTOS was motivated by the need for a language with a high abstraction level and a strong mathematical basis, suitable for the specification and analysis of complex systems. LOTOS consists of two integrated sub-languages for specifying data types (ADT – Abstract Data Types) and behaviour (process algebra). LOTOS has been used to specify and analyse a variety of systems. Many of these have been communications standards, but LOTOS has been successfully used in a number of other fields. LOTOS is supported by tools for specification, simulation, compilation, test generation and formal verification. LOTOS toolsets include CADP (CÆSAR/ALDÉBARAN Development Package [4]), LITE (LOTOS Integrated Tool Environment) and LOLA (LOTOS Laboratory). More information about LOTOS, tools, applications and publications can be found online [19].

Although LOTOS has proved to be widely applicable, ISO has been developing a revised version called E-LOTOS (Enhancements to LOTOS [8]). New language features of particular relevance to the invoicing case study include modularity, functional (constructive) data types, classical programming constructs, a controlled imperative style and strongly typed gates. Since E-LOTOS standardisation is ongoing, the authors have used a snapshot of the language.

In LOTOS and E-LOTOS, a system (the entity being specified) is modelled as one or more processes that communicate with each other and with their environment (whatever is outside a process, e.g. its user). The communication ports of a process are called (event) gates. LOTOS processes are parameterised by their gates and the state information they maintain. Inputs and outputs correspond to LOTOS events, i.e. interactions at a gate between two processes such as the system and its environment. It will be seen later that the inputs of Fig. 2.1 correspond to event offers made in the specifications. In LOTOS, an event occurs when two parties synchronise on matching event offers. An event offer indicates a willingness to communicate at a gate. Since several events may

be offered, a choice may have to be made of which event offers are synchronised and therefore which event actually occurs. This choice may affect the future behaviour of the system.

2.1.2 Requirements Capture in LOTOS

LOTOS is often used to specify 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. [2,15], 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.

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. Indeed it might be fairly said that LOTOS specifiers are pre-occupied with specification style! The choice of 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 of a system. (In the sense of [17], the architecture of a specification means its structure and style.) Several LOTOS workers have considered general 'quality' principles for specification architecture [12,17].

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 [9]. 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.

The case study treated in this book is data-oriented in nature 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 were used by the authors. These give some idea of the range of styles open to the LOTOS specifier.

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)?

The case study deals with requirements capture, analysis, specification and verification of the invoicing system. Of necessity this chapter presents only an overview of the specifications and their verification. Full details can be found in [14]. 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 provide answers in a sensible fashion. Analysis was performed according to the method outlined above. Some answers (**Answer** in the following) came from a common-sense reading of the requirements. Others (**Answer+**) required interpretation or extension of the requirements. As will be seen, the volume of questions is much greater than the informal problem statement! Of course, this demonstrates the value of a formal method.

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, variables in lower case, other identifiers with an initial capital).

2.2 Analysis and Specification of Case 1

The first case is discussed in this section though, as will be seen, it is treated as a simple abstraction of the second case.

2.2.1 Analysis

Methodology

Question 1: *Is Case 1 a simplification/abstraction of Case 2? Is Case 2 an extension/refinement of Case 1?*

Answer+: *It is not clear what the relationship between the cases is meant to be. In the authors' opinion, Case 1 does not make sense in isolation from Case 2. Note that this is a methodological issue, not a* LOTOS *issue. From the* LOTOS *point of view, Case 1 could have been specified without reference to Case 2. Orders cannot realistically be satisfied from stock under all circumstances, even if the informal problem statement permits this assumption. Sometimes it is better if the analyst does not literally accept everything the client says! The system could be placed in an impossible situation if the assumption were violated. The informal description of Case 1 also supposes some unidentified agency that maintains orders and stocks. For both these reasons, the authors do not regard Case 1 as sensible in its own right – only as a simplification of Case 2. It was therefore decided to treat Case 2 as primary, with Case 1 being an abstraction of this. The analysis in this section is therefore confined to those questions that arise from Case 1 alone.*

Interfaces

Question 2: *What does being 'given stock and the set of orders' imply?*

Answer: *This means that the first case study deals with a closed system that does not directly accept stock or order changes. 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.*

Functionality

Question 3: *Why is it said that all ordered references are in stock?*

Answer+: *This is presumably a hint that stock levels should not be checked before an order is invoiced. However it is not a realistic assumption. It is therefore prudent to check stock levels in this case, even if the check proves to be redundant.*

Question 4: *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. How should such apparently contradictory requirements be resolved?*

Answer+: *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! This interpretation does not directly affect the specification, but is a necessary stage in understanding the requirements.*

2.2.2 Specification

Case 1 is viewed as an abstraction of Case 2. There is therefore no externally observable behaviour since the system is closed. For Case 2, the process-oriented style introduces

some internal structure to the specification. The structure of the specifications to be presented is pictured in Fig. 2.1. The outer ovals in this figure represent the boundaries of alternative models. These correspond to the specifications for Case 1, Case 2 in data-oriented form, and Case 2 in process-oriented form. In each specification, the inner details are hidden from external view. There are thus three alternative levels of abstraction. 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).

Fig. 2.1. Structure of The Alternative Models/Specifications

Since Case 1 is treated as a simple modification of Case 2, the primary specifications are given in Sect. 2.3. Note that this was a modelling choice and is not intrinsic to how LOTOS might have been used. In fact the modifications are trivial, requiring only the internal communication channels to be hidden. For example, the process-oriented E-LOTOS specification has the following added:

```
hide Request:(Reference, Product, Amount)          (* hide internal gates *)
  Cancel:Reference, Deposit:(Product, Amount),
    Withdraw:(Product, Amount) in
```

2.3 Analysis and Specification of Case 2

The meat of this chapter lies in analysing and specifying the second case. Four specifications will be presented, using E-LOTOS and LOTOS in process-oriented and data-oriented styles.

2.3.1 Analysis

Environment

Question 5: *Is it necessary to know how many users there are?*
Answer: *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. produce an invoice or deliver a product), this question is purely for understanding the requirements. It does not affect the* LOTOS *specifications directly.*

Interfaces

Question 6: *At what point is the status of an order updated? If the status is updated when the stock or set of orders changes, how does the system know that there has been a change? If the status is updated following a periodic check, how frequently should the system check?*
Answer+: *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.*
Question 7: *Are new stocks or orders notified individually or in batches to the system?*
Answer+: *For simplicity it is assumed that inputs occur individually.*
Question 8: *How is invoicing triggered, how is the information obtained, and how is the decision made to update orders?*
Answer+: *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.*
Question 9: *How is it possible to identify an order to be cancelled?*
Answer+: *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 would be redundant on cancellation and so are omitted.*
Question 10: *Who, then, is responsible for creating an order reference?*
Answer+: *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

Question 11: *What is the meaning of being able to 'change the state of an order' ?*
Answer: *It is presumed that the system merely inspects the state of current orders and adjusts their status according to the current stock.*
Question 12: *Should the system issue an invoice when 'changing the state of an order' ?*
Answer: *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 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.*
Question 13: *What is implied by the system changing the state of an order from 'pending' to 'invoiced'?*
Answer+: *It is not clear whether this means that orders should be explicitly associated with a status. Presumably so, though the status of an order might be implicit (e.g. if unfulfilled orders are held separately).*
Question 14: *It is said that several orders may cite the same product code. This seems an almost unnecessary remark, but it hints that several orders may be outstanding for the same product. In this case, how should stock be allocated to orders?*
Answer+: *The stock is limited by implication, 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. Specifically, the allocation algorithm is not visible to or influenced by the system environment, i.e. it is non-deterministic.*
Question 15: *If an order can be fulfilled from stock, its state must be changed to 'invoiced'. What should happen if an order cannot be fulfilled because the stock is insufficient?*
Answer+: *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.*
Question 16: *What should be done if an order is held until stock becomes available?*
Answer+: *When the system is given new stock it must re-examine any unfulfilled orders to see if they can be satisfied. As discussed in Question 14, there is then an issue of how stock should be allocated. Again, a 'random' algorithm is assumed.*
Question 17: *What does cancelling an order mean?*
Answer+: *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. Cancelling a non-existent or invoiced order is assumed to be forbidden.

Question 18: *Is any concurrent or distributed processing required?*

Answer+: *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

Question 19: *Is it required that an order carry a product code?*

Answer: *This is implicit.*

Question 20: *Is there any restriction on the product quantity carried by an order?*

Answer+: *Presumably the quantity must be a positive integer. Negative quantities might correspond to returned products. A zero-quantity order is conceivable, but 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 for simplicity this was not allowed. Similarly, stock deposits are assumed to be strictly positive integers.*

Question 21: *Are there any orders or stocks initially?*

Answer+: *This is not explicitly stated in the informal requirements. Conceivably there could be an initial setup of orders or stocks, but for simplicity it is presumed that these are empty at start-up.*

Non-Functional Properties

Question 22: *As would normally be expected, are there any non-functional requirements such as cost, delivery schedule, performance, reliability, integration and testing?*

Answer: *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.*

Question 23: *Must an uncancelled order be satisfied eventually?*

Answer: *It is assumed that the system behaves fairly, and does not indefinitely delay the processing of an order.*

Question 24: *Must the system be free from deadlocks?*

Answer: *This requirement is implicit, but leads to a formal property of the specification that should be checked.*

Question 25: *Must the system be free from livelocks (i.e. unbounded loops of internal actions)?*

Answer: *This requirement is implicit, but leads to a formal property of the specification that should be checked.*

2.3.2 Specification

Since E-LOTOS is the future form of the language, the authors felt it would be interesting to see how its specification style differed from LOTOS. Both E-LOTOS and LOTOS specifications of the case study have therefore been prepared. Since the languages differ, each has been written in its native style; the specifications are not just syntactic translations of each other. There are thus eight specifications in total, corresponding to $\{\text{Case 1, Case 2}\}\times\{\text{process-oriented, data-oriented}\}\times\{\text{E-LOTOS, LOTOS}\}$. For space reasons, these have not been presented in full.

Process-Oriented Specifications

Process-Oriented E-LOTOS Specification Gates in E-LOTOS are typed, allowing static checks on the kinds of values that are communicated. E-LOTOS event offers such as *Request(!ref, ?prd, ?amt)* may be synchronised (matched) with others. A fixed value in an event offer is preceded by '!', whereas a value to be determined in an event offer is preceded by '?'. These notations are also used in pattern-matching of expressions. Although '!' and '?' in event offers can usually be interpreted as output and input respectively, there is technically no distinction between these in LOTOS. This is because '?' is just a shorthand way of offering all the values from a set (e.g. '*?prd*' means 'offer any product reference').

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 that object, out of the whole collection, to synchronise on the messages intended for it. The specification allows several pending orders to compete simultaneously for the same product (whose stock levels may not be sufficient to satisfy all the orders). Since these orders are handled concurrently, the sequence in which they are satisfied is non-deterministic.

The data types and processes are specified here in a separate module *OrderStock* for convenience. As in normal software engineering practice, an E-LOTOS module is a reusable and self-contained collection of definitions. Modules are maintained separately from the specification proper, that describes the whole system.

For clarity, separate types are introduced for an order *Reference*, a *Product* code, and a product *Amount*. For simplicity, these types simply rename the natural number type (non-negative integers); library types like this can be used without explicit importing. If desired, structured types could be introduced for order references and product codes.

```
module OrderStock is                          (* order-stock definitions *)
  type Reference renames Nat endtype          (* order references *)
  type Product renames Nat endtype            (* product codes *)
  type Amount renames Nat endtype             (* product amounts *)
```

The status of an order is defined using an enumerated type. Orders start out blank, i.e. their product and amount have not yet been defined. Such an order is said to have status *None*. The complete type for an order is given as a record containing product, amount and status fields.

```
type Status is enum None, Pending, Invoiced endtype      (* order status *)
type Order is                                            (* order *)
  record Prod:Product, Amt:Amount, Stat:Status
endtype
```

A blank order is filled in when an *Order* object accepts an order *Request*. A pending order may accept a *Cancel* and be annulled. A pending order may also *Withdraw* from stock. A choice is made from these possibilities using the '[]' (choice) operator that offers alternative behaviours. One of these is selected by matching event offers with the environment of the *Order* object. Other common LOTOS operators include ':=' (for assignment) and ';' (for sequential behaviour).

An event offer may be qualified by a condition (written in brackets after the offer). For example, a new order is permitted only if the order is blank (status *None*) and the amount being ordered is positive (*amt* > *0*); the order status then becomes *Pending*. Cancellation is allowed only if the order is pending, at which point the order ceases to exist (i.e. its status becomes *None*). 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]        (* order object gates, ... *)
 (ref:Reference, prd:Product, amt:Amount, sta:Status) is    (* parameters *)
  loop                                                     (* loop indefinitely *)
    Request(!ref, ?prd, ?amt)        (* new order if blank, amount positive *)
      [(sta == None) and (amt > 0)];
    ?sta := Pending                                        (* set status pending *)
  []                                                       (* or *)
    Cancel(!ref) [sta == Pending];               (* cancel order if pending *)
    ?sta := None                                 (* set status not in use *)
  []                                                       (* or *)
    Withdraw(!prd, !amt)          (* withdraw product for pending order *)
      [sta == Pending];
    ?sta := Invoiced                             (* set status invoiced *)
  endloop                                                  (* end main loop *)
endproc                                                    (* end order object *)
```

A *Stock* object repeatedly accepts deposits from the environment and withdrawals from order objects. New stock (of positive amount) is added to the current stock-holding. Withdrawal is permitted if the requested amount can be taken from current stock.

```
process Stock [Deposit, Withdraw]                (* stock object gates, ... *)
 (prd:Product, amt:Amount) is                                (* parameters *)
  var newamt:Amount in                          (* variable declarations *)
    loop                                                   (* loop indefinitely *)
      Deposit(!prd, ?newamt)          (* deposit stock if amount positive *)
        [newamt > 0];
      ?amt := amt + newamt                     (* increase stock amount *)
```

```
    []                                                          (* or *)
      Withdraw(!prd, ?newamt)   (* withdraw stock if sufficient amount *)
       [newamt <= amt];
      ?amt := amt − newamt                  (* decrease stock amount *)
    endloop                                          (* end main loop *)
  endvar                                        (* end variable scope *)
endproc                                           (* end stock object *)
```

The module concludes by defining unbounded sets of processes for *Orders* and *Stocks*, each running independently in parallel (denoted '|||'). 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]          (* orders gates, ... *)
 (ref:Reference) is                                        (* parameter *)
  Order [Request, Cancel, Withdraw] (ref, 0, 0, None)     (* one order *)
|||                                                   (* in parallel with *)
  Orders [Request, Cancel, Withdraw] (ref + 1)          (* more orders *)
endproc                                                  (* end orders *)
process Stocks [Deposit, Withdraw]                  (* stocks gates, ... *)
 (prd:Product) is                                          (* parameter *)
  Stock [Deposit, Withdraw] (prd, 0)                    (* one stock item *)
|||                                                   (* in parallel with *)
  Stocks [Deposit, Withdraw] (Succ(prd))              (* more stock items *)
endproc                                                  (* end stocks *)
endmod                                  (* end order-stock definitions *)
```

The overall specification 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     (* use order-stock module *)
  gates Request:(Reference, Product, Amount),      (* gates and their types *)
   Cancel:Reference, Deposit:(Product, Amount)
  behaviour                                      (* specification behaviour *)
```

Communication between orders and stocks is via an internal gate *Withdraw* (see Fig. 2.1). The order and stock processes synchronise on withdrawal. An operator such as '|[*Withdraw*]|' names the gates on which parallel behaviours must synchronise. The first order reference and product code are given as 0. As new orders and stocks arrive, the processes will update their state and will communicate to satisfy orders.

```
    hide Withdraw:(Product, Amount) in     (* hide internal withdraw gate *)
      Orders [Request, Cancel, Withdraw] (0)                 (* orders *)
    |[Withdraw]|                 (* synchronising on withdrawals with *)
      Stocks [Deposit, Withdraw] (0)                         (* stocks *)
endspec                                          (* end specification *)
```

Process-Oriented LOTOS Specification The equivalent process-oriented specification in LOTOS is similar, except that modules are not available. The specification is self-contained behaviour that continues indefinitely ('**noexit**'). Natural numbers are selected from the standard library. Since this type does not define subtraction, a definition of this is given (though not here, for brevity).

```
specification Invoicing [Request, Cancel, Deposit] : noexit          (* gates *)
  library NaturalNumber endlib                                       (* use naturals in library *)
```

As for the E-LOTOS process-oriented specification, order references, product codes and product amounts are specified by renaming naturals. These types, along with the similar status type, are omitted here. The overall behaviour is also left out as it is similar to the E-LOTOS version. 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]          (* order object gates, ... *)
 (ref:Reference, prd:Product, amt:Amount, sta:Status) : noexit :=  (* pars *)
  [sta = None] ->                                   (* blank order status? *)
    Request !ref ?prd:Product ?amt:Amount [amt gt 0];
                                    (* order request for positive amount *)
    Order [Request, Cancel, Withdraw]               (* set pending status *)
     (ref, prd, amt, Pending)
[]                                                  (* or *)
  [sta = Pending] ->                                (* pending order status? *)
    (
      Cancel !ref;                                  (* cancel order *)
      Order [Request, Cancel, Withdraw]             (* set blank status *)
       (ref, 0 of Product, 0 of Amount, None)
    []                                              (* or *)
      Withdraw !prd !amt;                           (* withdraw stock *)
      Order [Request, Cancel, Withdraw]             (* set invoiced status *)
       (ref, prd, amt, Invoiced)
    )
endproc                                             (* end order object *)
process Stock [Deposit, Withdraw]                   (* stock object gates, ... *)
 (prd:Product, amt:Amount) : noexit :=              (* parameters *)
  Deposit !prd ?newamt:Amount                       (* deposit if positive amount *)
   [newamt gt 0];
  Stock [Deposit, Withdraw]                         (* repeat with increased stock *)
   (prd, amt + newamt)
[]                                                  (* or *)
  Withdraw !prd ?newamt:Amount                      (* withdrawal if sufficient amount *)
   [newamt le amt];
  Stock [Deposit, Withdraw]                         (* repeat with decreased stock *)
   (prd, amt − newamt)
endproc                                             (* end stock object *)
```

```
endspec                                                    (* end specification *)
```

Data-Oriented Specifications

Data-Oriented E-LOTOS Specification In this approach, orders and stocks are defined by data values rather than processes. Invoicing is then an operation on these values. The data types are not given here since they closely resemble the process-oriented E-LOTOS ones. 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. An array element is accessed by operation *Get* and stored by *Put*. Similar operations are used with record fields, e.g. *Get_Stat* and *Set_Stat* for the order status field.

Invoicing orders is performed by function *Invoice* 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, whereas the process-oriented specification deals with them non-deterministically. 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) :    (* invoicing parameters, ... *)
 (Orders, Stocks) is                                              (* results *)
  var ref:Reference, prd:Product, amt,                (* variable declarations *)
   stk:Amount, sta:Status in
    for (?ref := First(ords); ref <= Last(ords);                (* for all orders *)
     ?ref := Next(ords, ref)) do
      (?prd, ?amt, ?sta) := Get(ords, ref);                 (* get order details *)
      ?stk := Get(stks, prd);                            (* get stock for product *)
      if (sta == Pending) and (amt <= stk) then      (* pending, in stock? *)
        ?ords :=                                          (* set order invoiced *)
          Put(ords, ref, Set_Stat(Get(ords, ref), Invoiced));
        ?stks := Put(stks, prd, stk − amt)             (* decrease stock level *)
      endif                                        (* end pending order check *)
    endfor                                                  (* end order loop *)
    (ords, stks)                                (* return resulting orders, stocks *)
  endvar                                                  (* end variable scope *)
 endfunc                                                (* end invoicing function *)
endmod                                                (* end order-stock definitions *)
```

The system specification is like that for the process-oriented E-LOTOS version except that local variables are introduced for orders, stocks, order reference and product

code. Orders and stocks are initialised to be empty. The main behaviour repeatedly accepts order requests, order cancellations and stock deposits. The logic is as already seen, except that the 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 deal with pending orders and to alter stocks accordingly.

```
        loop                                                    (* loop indefinitely *)
          (                                          (* choice of request, cancel, deposit *)
            Request(?ref, ?prd, ?amt)              (* new order, positive amount? *)
              [NotIn(ords, ref) and (amt > 0)];
            ?ords :=                               (* update orders with pending order *)
              Put(ords, ref, Order(prd, amt, Pending))
          []                                                                    (* or *)
            Cancel(?ref)                           (* cancel order if exists and pending*)
              [IsIn(ords, ref) andthen (Get_Stat(Get(ords, ref)) == Pending)];
            ?ords := Delete(ords, ref)                                  (* delete order *)
          []                                                                    (* or *)
            Deposit(?prd, ?amt) [amt > 0];         (* deposit positive amount *)
            ?stks := Put(stks, prd,    (* update stocks with extra/new amount *)
              if IsIn(stks, prd) then Get(stks, prd) + amt else amt endif)
          );                                                        (* end choice *)
          (?ords, ?stks) := Invoice(ords,stks)       (* get new orders, stocks *)
        endloop                                                  (* end main loop *)
      endvar                                                    (* end variable scope *)
    endspec                                                     (* end specification *)
```

Data-Oriented LOTOS Specification This 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 that maps values to the natural numbers. 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, *MkOrder* and *MkStock* operations are needed.

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. (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.) *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. 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                                (* order list *)
```

```
sorts Orders                                          (* name for order list *)
opns                                                  (* operations *)
  NoOrders : -> Orders                                (* empty list of orders *)
  AddOrder : Order, Orders -> Orders                  (* add order to order list *)
  RemOrder : Order, Orders -> Orders (* remove order from order list *)
  StatOrder : Reference, Orders -> Status             (* get status for reference *)
eqns                                                  (* equations *)
  forall ref1,ref2:Reference, prd1,prd2:Product,      (* globals *)
   amt1,amt2:Amount, sta1,sta2:Status, ords:Orders
    ofsort Status                                     (* operations yielding status *)
      StatOrder(ref1, NoOrders) = None;               (* no orders, no status *)
      ref1 eq ref2 =>                                 (* order references match? *)
        StatOrder(ref1,                               (* get status *)
          AddOrder(MkOrder(ref2, prd2, amt2, sta2), ords)) = sta2;
      ref1 ne ref2 =>                                 (* order references differ? *)
        StatOrder(ref1,                               (* check other orders *)
          AddOrder(MkOrder(ref2, prd2, amt2, sta2), ords)) =
            StatOrder(ref1, ords);
    ofsort Orders                                     (* operations yielding orders *)
      ref1 eq ref2 =>                                 (* order references match? *)
        RemOrder(                                     (* remove order from list *)
          MkOrder(ref1, prd1, amt1, sta1),
            AddOrder(MkOrder(ref2, prd2, amt2, sta2), ords)) = ords;
      ref1 ne ref2 =>                                 (* order references differ? *)
        RemOrder(       (* remove order from list by checking others *)
          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                                               (* end order list *)
```

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). As has just been seen, LOTOS data types are reasonably straightforward but lengthy. The *Stocks* type is therefore omitted here.

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. (As in 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                        (* order-stock updates *)
  opns                                                (* operations *)
    UpdateOrders : Orders, Stocks -> Orders           (* update orders *)
```

```
    UpdateStocks : Orders, Stocks -> Stocks              (* update stocks *)
  eqns                                                    (* equations *)
    forall ref:Reference, prd:Product, amt:Amount,        (* globals *)
     sta:Status, ords:Orders, stks:Stocks
      ofsort Orders                          (* operations yielding orders *)
        UpdateOrders(NoOrders, stks) =  (* no orders, no orders update *)
          NoOrders;
        (sta eq Pending) and (StockOf(prd, stks) ge amt) =>
                                          (* pending order, sufficient stock? *)
          UpdateOrders(       (* update orders by setting order invoiced *)
            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) =>
                                            (* invoiced or insufficient stock? *)
          UpdateOrders(          (* update orders by leaving order alone *)
            AddOrder(MkOrder(ref, prd, amt, sta), ords), stks) =
              AddOrder(MkOrder(ref, prd, amt, sta),
                UpdateOrders(ords, stks));
      ofsort Stocks                          (* operations yielding stocks *)
        UpdateStocks(NoOrders, stks) =    (* no orders, no stock update *)
          stks;
        (sta eq Pending) and (StockOf(prd, stks) ge amt) =>
                                          (* pending order, sufficient stock? *)
          UpdateStocks(             (* update stocks by decreasing stock *)
            AddOrder(MkOrder(ref, prd, amt, sta), ords), stks) =
              UpdateStocks(ords, RemStock(MkStock(prd, amt), stks));
        (sta eq Invoiced) or (StockOf(prd, stks) lt amt) =>
                                            (* invoiced or insufficient stock? *)
          UpdateStocks(          (* update stocks by leaving stock alone *)
            AddOrder(MkOrder(ref, prd, amt, sta), ords), stks) =
              UpdateStocks(ords, stks);
  endtype                                          (* end order-stock updates *)
```

The overall behaviour is similar to that for the E-LOTOS data-oriented specification, though the syntax is different. Explicit recursion is used to express a loop. Each branch of the choice produces an updated pair of order-stock values. A recursive call to the *Invoice* process updates the orders and stocks following invoicing.

2.4 Validation and Verification of the LOTOS Specifications

The terms 'validation' and 'verification' are used with various meanings in the literature. The authors use these terms to mean testing and proof respectively. Since E-LOTOS has not yet reached its final form, tools for the language are still under development. The E-LOTOS specifications should hence be regarded as conceptual at this stage, although

they have been statically checked using the TRAIAN tool. However they are similar to the LOTOS specifications and have been independently reviewed, so there is a high degree of confidence in them. The following discussion therefore refers to automated analysis of only the LOTOS specifications.

2.4.1 Validation

The LOTOS specifications were initially validated using the LITE toolset 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. For example, *RemOrder* was checked with an empty list of orders (*NoOrders*) and a list containing at least one order (*RemOrder(SomeOrder, MoreOrders)*). The latter has two sub-cases: the order reference exists in the list of orders, and the order reference does not exist.

Behavioural aspects 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 executing the scenarios and recording the reactions of the specified system. Normally the client would be asked to confirm the completeness and correctness of testing, but that was not possible in this case study. Instead the authors reviewed the behaviour exhibited by the specifications. This uncovered some small specification errors, notably in the LOTOS data types (which are rather complex).

2.4.2 Verification

Model Generation Validation cannot ensure the correctness of the specification. Neither is it possible to prove equivalence between various specifications. To achieve this requires formal verification, for which the authors used model-checking [3] and equivalence-checking [5]. These procedures are automated, but apply only to a system with a finite state space.

CADP (CÆSAR/ALDÉBARAN Development Package [20]) provides several tools for the design and verification of communications protocols and distributed systems. The CADP tools used in the case study will be mentioned only briefly. CÆSAR and CÆSAR.ADT are compilers that translate a LOTOS specification into a (possibly infinite) LTS (Labelled Transition System) that encodes all possible execution sequences. ALDÉBARAN is a verification tool for comparing or minimizing LTSs with respect to various equivalence relations [10]. XTL (Executable Temporal Language) is a functional-like programming language for compact implementation of various temporal logics.

An LTS is formally defined as a quadruple $M = \langle Q, A, T, q_{init} \rangle$. Q is the set of states, A is the set of actions, and $T \subseteq Q \times A \times Q$ is the transition relation between the states. A transition $\langle q_1, a, q_2 \rangle \in T$ (also written $q_1 \xrightarrow{a} q_2$) means that it is possible to move from state q_1 to state q_2 by performing action a. State $q_{init} \in Q$ is the initial one.

The translation of a LOTOS specification into an LTS respects the operational semantics of LOTOS. A state of the LTS represents a possible behaviour of the specification, and each transition from a state is labelled with a possible action of the behaviour.

To compile data operations efficiently, CÆSAR.ADT needs to know which operations are the constructors (i.e. the primary operations that build values). Also, data type equations are considered as rewriting rules (oriented from left to right), and equations between constructors are not allowed. For tool use, the data types are therefore annotated and some transformations are applied [6].

To ensure finiteness, the domains of various parameters were restricted for verification purposes. Specifically, an upper bound was set on the highest value for order references, product codes and order amounts (denoted *MaxRef*, *MaxProd*, and *MaxAmt*). An infinite number of parallel processes is implied by the specifications since the number of orders and stock items is unlimited. A specific number of process instances was used according to the upper bound chosen for the parameters. As a practical limitation, only certain combinations of parameter bounds were investigated: *MaxRef* values of 1 or 2, *MaxProd* values of 0 and 1, *MaxAmt* values of 1 and 2. In principle this generates 8 LTSs for each specification under consideration.

Verification requires knowledge of all the actions performed by the system. Specifications of Case 2 can be used directly. However, those for Case 1 cannot because visible actions are deliberately made internal. Models are therefore generated only for Case 2. Execution times in Tables 2.1 and 2.2 were obtained when using the CADP tools on a SUN computer (Ultra Sparc-1, 143 MHz processor, 256 MBytes memory).

Model generation for the process-oriented specification of Case 2 is summarised in Table 2.1. This is limited by state explosion. The three last rows of the table indicate the incomplete LTSs generated before memory is exhausted. The reason for the state explosion is the high degree of interleaving in this specification. Note that the LTS size increases by one order of magnitude when *MaxRef* or *MaxAmt* is incremented. Moreover, it increases more sharply with *MaxAmt* than with *MaxRef*.

Table 2.1. Model Generation for the Process-Oriented Specification of Case 2

MaxProd	*MaxRef*	*MaxAmt*	States	Transitions	Time (mins.)
0	1	1	5,890	16,130	0.1
0	1	2	39,371	170,754	0.5
0	2	1	25,846	96,430	0.4
0	2	2	323,459	1,826,512	5.9
1	1	1	7,698,453	35,655,750	132.8
1	1	2	$>$ 6,531,532	$>$ 49,232,000	164.5
1	2	1	$>$ 8,213,739	$>$ 49,896,000	383.0
1	2	2	$>$ 4,524,531	$>$ 43,904,000	237.1

Model generation for the data-oriented specification of Case 2 is summarised in Table 2.2. The limit here is execution time not memory occupancy: the last row of the table stays within memory limits but takes an inordinate amount of time. The reason is

that interleaving in the process-oriented specification is replaced by data computations. The functions used by the data-oriented specification perform several traversals over the lists containing the orders and the stocks, and these are relatively complex structures. It is interesting to note that the LTS size increases by one order of magnitude when either *MaxRef* or *MaxAmt* is incremented. In the data-oriented case the model size increases more sharply with *MaxRef* than with *MaxAmt*.

Table 2.2. Model Generation for the Data-Oriented Specification of Case 2

MaxProd	*MaxRef*	*MaxAmt*	States	Transitions	Time (mins.)
0	1	1	14,975	20,195	0.5
0	1	2	88,023	165,792	4.0
0	2	1	82,403	117,386	5.6
0	2	2	848,067	1,603,478	165.3
1	1	1	$>$ 5,236,886	$>$ 9,761,401	31935.4

Verification using Model Checking The generated LTSs are verified against formal properties stated in XTL. Since the informal requirements do not state formal properties, these have to be inferred. As the dynamic semantics of LOTOS is event-based, it is natural to express the properties as temporal logic formulae concerning actions. The XTL language supports data types for states, transitions and labels. It also supports functions for manipulating them.

The informal requirements were interpreted as described in Sects. 2.2.1 and 2.3.1. The new requirements can be split into three (overlapping) classes: those that cannot be formalised, those are self-evidently reflected in the specifications, and those that lead to formal properties. The last of these requirements were formalised as safety and liveness properties. Safety properties state that something bad never happens, while liveness properties state that something good eventually happens.

For conciseness, only the first property below is presented in full detail. This gives an informal statement, a more precise formulation (in italics), a formalisation in XTL, and an explanation of this. For the other properties, only the informal statement and the precise formulation (in italics) are given. The full formulation of these properties may be found in [14].

Property 1: A safety property is that the quantity carried by an order must be a positive integer. *All Requests have strictly positive amounts*.

not exists L : **label in**
L $\rightarrow$ [Request _ _ ?amt : **integer where** amt $\leq$ 0]
end_exists

The **exists** operator checks here for existence of a label in the LTS corresponding to a *Request* with an amount that is not positive. Of course, such a label should not exist. Formulae are written in brackets as above. XTL is able to access the parameters of an event; the '_' notation stands for any value of event parameter.

Property 2: A safety property is that an order reference must be unique. Duplicate references to an order (i.e. more than one *Request* with the same reference) are allowed only if the order has not been invoiced (i.e. a *Cancel* for the order reference has been accepted). *Between two subsequent Requests with the same reference* $(0 \leq ref \leq MaxRef - 1)$*, a Cancel action with the same reference must occur.*

Property 3: A safety property is that only existing orders may be cancelled. *A Cancel action with some reference* $(0 \leq ref \leq MaxRef - 1)$ *can appear only if there has been a Request with the same reference*.

Property 4: A liveness property is that an uncancelled order will eventually be invoiced. Note that this property makes a statement about the state variables of the specifications (i.e. the status of an order). Since the underlying model of LOTOS is an LTS, a state variable cannot be checked directly. Instead, only properties over transitions can be verified. It is therefore necessary to introduce an explicit *Invoice* event that notes when invoicing occurs. *After a Request with some reference* $(0 \leq ref \leq MaxRef - 1)$*, if a Cancel with the same reference does not occur then eventually an Invoice action for the reference will occur.*

Property 5: System behaviour should always progress. *The system is free from deadlock.*

Property 6: System behaviour should not get stuck in an internal loop. The need for this property arises because of internal actions in the specifications. These arise from hiding gates (*Withdraw*) or indirectly as a result of the enabling operator ($\gg$). *The system does not livelock, i.e. there are no cycles of internal actions*.

The six properties were evaluated on the complete LTS models using the XTL model-checker. The models were first minimised using strong equivalence in order to reduce their size. In every case the properties were verified in less than one minute. Properties 1, 2, 3, 5 and 6 are true of all the models. With the introduction of an *Invoice* event to verify property 4, all the properties were shown to hold.

Verification using Bisimulation As an alternative to model-checking, verification using bisimulation was also performed using ALDÉBARAN. The goal was to prove that the process-oriented and data-oriented LOTOS specifications of Case 2 are equivalent by checking their LTSs. ALDÉBARAN supports several notions of equivalence, ranging from strong bisimulation equivalence (each specification can mimic the other) to safety equivalence (neither specification violates safety properties of the other). The ALDÉBARAN tool was used to check all the possible forms of equivalence. Branching equivalence and observational equivalence cannot be checked since the specifications contain internal events.

It was found that the data-oriented specification is included in the process-oriented one, but not equivalent to it when using safety equivalence. This equivalence does not hold because the process-oriented specifications allow *Cancel* events after *Deposit* events if the order is not yet treated. In the data-oriented specification this cannot occur since the list of orders is immediately updated after each *Deposit* event.

This difference can be removed by not immediately updating the stocks and orders in the the data-oriented specification. Instead, an internal event is required before

updating can occur. This has the positive effect of making the data-oriented and process-oriented specifications safety equivalent. Unfortunately it also introduces livelock due to continual updating of stocks and orders. Although this violates Property 6, a more complex modification of the specification would avoid livelock.

2.5 Natural Language Description of The Specifications

The following summaries describe how the two cases have been interpreted in the (E-)LOTOS specifications.

2.5.1 Case 1

As discussed under Question 1, Case 1 is considered to be an abstraction of Case 2. Its specifications merely hide (make internal) the actions that *Request* orders, *Cancel* orders, and *Withdraw* products from stock. This applies equally to the LOTOS/E-LOTOS and process-oriented/data-oriented variants.

2.5.2 Case 2

The E-LOTOS variants have separate modules for handling orders and stocks. This cleanly separates the subsidiary operations from the main specification. Modules do not exist in LOTOS, so specifications have to be written in isolation. For E-LOTOS and LOTOS, basic types are defined for an order *Reference*, a *Product* code, a product *Amount*, and an order *Status*. These are used to define the main type for an *Order*

In the process-oriented E-LOTOS specification, *Order* and *Stock* items are represented as processes. A blank *Order* may be filled in by a *Request*, a pending *Order* may be annulled by a *Cancel* or it may *Withdraw* stock. A *Stock* item may accept the *Deposit* of new stock, or may allow an *Order* to *Withdraw* stock. Sets of *Orders* and *Stocks* synchronise on *Withdraw* actions that are hidden from external view. The process-oriented LOTOS specification is broadly similar to its E-LOTOS counterpart, except that the main loop must be specified using explicit recursion.

In the data-oriented E-LOTOS specification, lists of *Orders* and *Stocks* are maintained as data values rather than as processes. An *Invoice* function is called to allocate pending orders to available stocks. The main behaviour simply loops, accepting *Request*, *Cancel* and *Deposit* actions. The data-oriented LOTOS specification is forced to use complex data types that achieve the effect of invoicing. Lists of *Orders* and *Stocks* are specified in data types with auxiliary operations on orders and stock items. The *Updates* type defines operations equivalent to the *Invoice* function. *UpdateOrders* yields the new order list following invoicing, while *UpdateStocks* does the same for the stock list.

2.6 Conclusion

This chapter has shown how (E-)LOTOS and LOTOS tools may be used for requirements analysis, formal specification and verification. Through the procedure of formal specification and verification, 25 questions were raised about the informal requirements. This in itself is an indication of the value of applying a formal technique.

The LOTOS approach of black-box specification is useful in obtaining a high-level formalisation of a system. Although LOTOS shares its behavioural approach with other process algebras, LOTOS is relatively unusual in having an integration of behaviour with data specification. This is convenient for specification since different aspects of a problem can be treated as behaviour or data. The process-oriented and data-oriented specifications show that these aspects can be balanced according to the system being specified.

Of the four Case 2 descriptions, the authors are 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 have been rather disliked for the verbosity that is evident from the specifications of the case study. 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 [11].

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 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 is thus closer to current analysis practice. The approach also hints at possible concurrent or distributed implementation, and thus may be closer to engineering practice.

Verification with CADP supports LOTOS with state-of-the-art techniques. As far as the authors know, CÆSAR and CÆSAR.ADT are the only model checkers that support dynamic data structures. It is useful to have the choice of verification through model-checking or equivalence-checking. Model-checking requires system properties to be explicitly formalised. It is not always obvious what these properties should be, nor whether 'enough' properties have been defined. Nonetheless, formulating system properties is a valuable exercise in its own right. For the case study, the properties highlighted some important issues that might otherwise have been overlooked.

The main difficulty with model checking is state-space explosion. This is, of course, not unique to LOTOS. To generate finite and tractable LTSs it is necessary to modify the original LOTOS specifications. Although these changes are relatively straightforward, they are nonetheless changes. The need to modify data types will disappear with E-LOTOS since this has constructive type definitions. However, it will continue to be necessary to limit the number of parallel processes considered during verification. Despite the restrictions imposed during verification, state-space explosion continued to be a problem. The source of this is the high degree of non-determinism in the process-oriented specifications and the complex data types in the data-oriented specifications. It

is hoped that ongoing development of CADP will help to minimise state-space explosion by producing reduced LTSs.

Equivalence checking was relatively straightforward, although a minor change still had to be made in the specifications to prove equivalence. However this reflects the different ways that the process-oriented and data-oriented specifications were written. The need for a change was thus for modelling rather than theoretical reasons.

Acknowledgements

Thanks are due to the following for carefully reviewing the papers that formed the basis of this chapter: Hubert Garavel (INRIA Rhône-Alpes), Radu Mateescu (INRIA Rhône-Alpes) and Carron Shankland (University of Stirling). Mihaela Sighireanu was employed by INRIA Rhône-Alpes during most of the work reported here.

References