Protocol Design and Implementation Using Formal Methods*

M. VAN SINDEREN, L. FERREIRA PIRES AND C. A. VISSERS
Tele-Informatics and Open Systems Group, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

This paper reports on a number of formal methods that support correct protocol design and implementation. These methods are placed in the framework of a design methodology for distributed systems that was studied and developed within the ESPRIT II Lotosphere project (2304). The paper focuses on design methods for synthesising protocols by successive application of correctness-preserving LOTOS transformations. This transformational approach is described in some detail and is illustrated with a protocol design example. The paper concludes with some suggestions for relating design methods to milestones in the protocol design and implementation processes.

Received June 1992

1. INTRODUCTION

The ESPRIT II Lotosphere project (2304) focused on a methodology for the rapid and correct design and implementation of distributed systems, in particular services and protocols, that should be suitable for industrial application. Correct design and implementation prompts the use of formal languages that support unambiguous and concise representation of designs. The standard Formal Description Technique (FDT) LOTOS12 has been adopted by the Lotosphere project because it fulfills the requirements for precision and conciseness in the representation of distributed system design concepts, such as interactions and system structure. Industrial applicability requires that tools supporting these design methods are made available. Design support tools produced in Lotosphere have been integrated in a tool environment called LITE (Lotosphere Integrated Tool Environment). By adding design methods and tools, the original gravity of LOTOS as a specification language was gradually turned into a design language.

This paper gives an overview of some design methods that have been developed in the Lotosphere project, focusing on a transformational approach to synthesising protocols from service specifications. This paper is further structured as follows: Section 2 presents an overview of the Lotosphere design methodology and its relevance to protocol design and implementation. Section 3 presents some transformations that can be used in protocol design, and Section 4 presents a concrete protocol design example to illustrate the transformational approach. Conclusions are drawn in Section 5.

2. THE LOTOSPHERE DESIGN METHODOLOGY

The Lotosphere design methodology adopts the usual view of a design trajectory as a sequence of design steps in which the user requirements are first formulated and then successively refined until the design of a real system is obtained. It further adapts this design trajectory on basis of notions such as rapid cycling using key functional elements, tree search, reference implementations, and the distinction between functional and non-functional properties of the system. 2,19

* This work has been partly supported by the CEC research programme ESPRIT II (Lotosphere, ref. 2304).

The methodology aims at using LOTOS for as long as possible throughout the design trajectory. This enables notions of implementation to be formalised, and design methods to be implemented in design support tools, all based on LOTOS. Fig. 1 depicts a simplified view of the Lotosphere design trajectory. Initially the user requirements are formally defined in terms of the observable behaviour of the system, such that only properties which are relevant to the system users are expressed. Successive intermediate designs represent design decisions concerning, for example, internal structure, data and behaviour refinements, and the mapping onto software and hardware structures.

In protocol design, the definition of the user requirements corresponds to a service specification, which is an integrated representation of the observable behaviour of a distributed system which can be experienced by the service users at service access points. The service abstracts away from all possible internal structures that can be given to the system. The service is used as the starting point for protocol design, in which this internal structure is gradually decided and defined.

Protocol design is usually carried out in two phases. In the first phase the functions of the protocol are defined as far as possible in an implementation-independent way, but in such a way that proper interworking of peer protocol entities is guaranteed. The resulting design supports the definition of open systems. Therefore it is usually produced (next to service design) by a (pre-) standardisation body, and published as an international standard or recommendation. The resulting design we call the protocol design, or simply the protocol. The next step in protocol design is its implementation-dependent design. During this step a protocol design is refined until a proper composition of (hardware and software) components is obtained that can be more easily mapped on to a real system. Such a composition we call the protocol implementation. Protocol designs and implementations must be correct with respect to the service, i.e. they must both implement the service.

LOTOS is a broad-spectrum design language allowing its to be used not only as a specification language for services and protocols (for which it was originally developed), but also as a language to represent intermediate protocol implementations. Formally defined LOTOS-to-LOTOS transformations support the design steps from the service to a protocol design and protocol implementation. These transformations can be repeated...
until we can compile a LOTOS specification directly or transform it by hand into code of a target implementation language (e.g. ADA or C).

The correctness of the real system can be checked by validating the user requirements and each refinement throughout the design trajectory. In addition, the real system can be tested for conformity to intermediate specifications.

The purpose of the next sub-sections is to present a taxonomy of design methods that support protocol design and implementation. Section 2.1 discusses the architectural and language considerations involved in a design step and defines the concept of design method, 2.2 presents a catalogue of LOTOS transformations to be used throughout a design process, 2.3 addresses protocol design and 2.4 tackles protocol implementation.

2.1 Architectural versus language considerations

Architectural concepts represent abstractions of aspects of technical objects of concern. In service and protocol design, for example, generic architectural concepts are service, service primitives, service access points, protocol, protocol entities, etc. A specification language serves the purpose of representing these architectural concepts. In case elements of a specification language (e.g. syntax and semantics elements) are derived from generic architectural concepts related to the technical area of concern, it may result in a general-purpose specification language.

An abstraction of a technical object of concern is called a design. A specification is a representation of a design using a specification language. This representation is necessary in the design process as a vehicle for analysis, communication, and manipulation, since the technical object itself is not necessarily available. This means that a distinction must be made between a design and its specification. This distinction is depicted in Fig. 2.

Design decisions taken during a design step must refine a design through the manipulation of architectural concepts. Since we can actually only manipulate specifications rather than the design itself, a design step is characterised by an architectural transformation, i.e. the conceptual modification of a design based on design objectives and including design decisions, and a language transformation, i.e. the manipulation of the specification of the design. There is not necessarily a one-to-one relationship between an architectural transformation and a language transformation. For example, in the design example presented in Section 4 we apply several language transformations to perform one architectural transformation.

We define the term architectural semantics as the relationship between architectural concepts and their possible representations in a specification language. More on architectural semantics for LOTOS can be found in Ref. 20.

We define a design method as a combination of language transformations used to perform an architectural transformation.

2.2 Catalogue of LOTOS transformations

Considering protocol design and implementation using LOTOS, several architectural transformations address problems that can be generalised and formalised in terms of general-purpose LOTOS-to-LOTOS transformations. The Lotosphere project produced a catalogue of these transformations; Table 1 presents a selection of the most prominent ones. Some of these transformations have
been implemented in tools of LITE. These are indicated with an * in Table 1. An overview of these tools can be found in Ref. 8.

In each transformation, some properties of an input specification are preserved and some others are modified in an output specification. In case the relationship between the input and output specifications can be formalised, we call it a correctness preserving transformation (CPT).

Since some or all semantics of the input specification is maintained in the output specification, designers must usually manipulate syntax elements in order to achieve a language transformation. An example is the replacement of one process in an input specification by multiple processes in the output specification in order to represent a structural refinement. Depending on the preserved properties of a transformation, different formally defined notions of correctness can be used. Some of these notions and associated laws can be found in Refs 11 and 15. In this paper we make use of two of these notions: observation equivalence ($\approx$), also called weak bi-simulation equivalence, and strong bi-simulation equivalence ($\sim$).

Some language transformations have solutions in terms of algorithms that, when applied to an input specification, produce a correct output specification. These algorithms can be fully or partly automated. Sections 3.1 and 3.2 present examples of algorithms that can be fully automated. Algorithms that can be only partly automated work interactively with the designer, by alternating between computation and request for more (design) information. Section 3.3 presents an example of an interactive algorithm.

### 2.3 Protocol design

In protocol design we assume that a formally defined initial service specification is available as an input specification and that we can focus on transforming this into a protocol specification. The iterative design strategy, based on repeated decomposition, can be used to achieve a final protocol specification. The decomposition process starts from an (N)-service specification and produces an (N)-protocol specification that is structured in terms of (N)-protocol entities and an underlying (N-1)-service. A protocol is a specific decomposition of the service, i.e., its specification includes specific design choices in the functions of the protocol entities and the underlying service. The underlying (N-1)-service is again decomposed in the same way as the (N)-service. This process is repeated until an underlying service is obtained that can be directly realised by an implementation component which matches the required behaviour. Fig. 3 depicts (layered) protocol design.

A service and a corresponding protocol are therefore alternative representations of the same distributed system behaviour, viewed from different perspectives. The service is an integrated representation of the system and is best specified in terms of a set of separate constraints on the occurrence of observable interactions (in OSI terminology service primitives). These constraints can be structured in terms of local constraints, i.e. constraints related to behaviour local to service access points, and remote or end-to-end constraints, i.e. constraints related to behaviour between different service access points. This form of representation is called constraint-oriented style.

A protocol is a distributed representation of a system, i.e. it represents the system in terms of a composition of objects or resources. Here a resource-oriented style is the most appropriate form for its specification. In a resource-oriented specification, the structure of the design is represented by a composition of processes, where a process represents an object or resource. The internal communication between resources is hidden from the environment of the system.

The protocol entities implement the distribution of the service. Therefore the local constraints of the service can reappear in the definition of the protocol entities, allowing re-use of earlier-defined elements of specification and facilitating validation. The remote constraints of the service can be decomposed over protocol entities and an underlying service. Specification styles are discussed in detail in Ref. 21. Fig. 4 depicts the use of specification styles in the formulation of service and protocol specifications.

### 2.3.1 Validation

One approach towards protocol design consists of designing a protocol using heuristics and using the service specification as a functional requirement, and validating the protocol afterwards against the service by analytical verification or by testing.

A verification method described in Ref. 22 assumes that the service and protocol specifications are structured in constraint-oriented and resource-oriented specification styles respectively. This choice of styles allows verification in a modular way. The method is exemplified by the
Protocol design and implementation using formal methods

Figure 4. Decomposition of a service into a protocol using specification styles.

verification of a protocol that provides a kind of question-answer service. If the underlying service comprises two independent directions of transfer, and the send and receive actions of each of the protocol entities are not related except by local constraints at the required service boundary, it turns out that modularisation is possible and that verification is relatively straightforward. Experience has shown, however, that in more realistic designs verification using analytical methods is not feasible.

Testing a protocol against the service, according to the formal notion of test equivalence, consists of defining test sequences derived from the service specification and applying them to the protocol specification. Test methods can be engineered, i.e. the ratio between the amount of computation necessary to achieve a certain coverage and the amount of computation to achieve complete coverage can be chosen. In most realistic designs, however, the use of testing does not constitute a formal proof of correctness.

2.3.2 Transformation

Another approach toward protocol design is derivation of a protocol specification from the service specification with the use of correctness preserving transformations. A design method that supports this approach is described in Ref. 9. The starting point of this method is a constraint-oriented service specification, structured in terms of local and remote constraints. It assumes that the service is point-to-point (there are only two service users), and that its specification has two local constraints (one for each of the service access points). The resulting specification contains components which can be considered as preliminary forms of the two desired protocol entities. These components interact directly with each other via synchronous interaction, which implies that the synchronous interaction must be refined, in terms of an underlying service.

In Section 4 we show design methods that allow the design of protocol entities that communicate using a reliable or an unreliable medium as underlying communication service, and thus go a step further than the design method described in Ref. 9.

2.4 Protocol Implementation

Some methods used for protocol design can be also used for protocol implementation. In addition, the Lotosphere project has addressed a couple of specific implementation approaches, of which the use of predefined implementation constructs and compilation represent two quite different alternatives.

2.4.1 Use of predefined implementation constructs

This approach assumes the availability of a set of high-level general-purpose implementation components, called predefined implementation constructs (PDICs). Each PDIC is represented by its formal specification in LOTOS, whereas a correct implementation (and realisation) of this formal specification in the target-implementation environment is available. The objective of using PDICs is to transform the protocol specification as early as possible in the design trajectory into a composition of PDIC specifications. Since the correctness
of each PDIC implementation has been already established by the time of its construction, the correctness of a protocol implementation becomes only dependent on the correctness of the composition of PDIC specifications. This can be established early in the design trajectory on basis of a formal relation between Lotos specifications, i.e. the formal specification of the protocol and its corresponding composition of PDIC specifications.

Using this approach, designers are encouraged to develop a library of general-purpose PDICs, such that many different protocols can be implemented using this library. Fig. 5 depicts this approach.

Figure 5. The PDICs approach.

The use of PDICs has the advantage of bringing a clear distinction between structural (architectural) design and the technicalities inherent to different implementation environments. It also has the potential of reducing validation efforts and shortening the design life cycle of individual protocols. The approach has the disadvantage that designers have to cope with the development and maintenance of a library of PDICs. Furthermore, automated methods to transform protocol specifications into compositions of PDICs are not (yet) available.

2.4.2 Compilation

Two compilers have been developed in Lotosphere and are available in LITE: TOPO and COLOS. TOPO is a tool that generates C code from a Lotos specification, by transforming the specification into a model called the A/P model.14 The Lotos specification is then implemented as a composition of behaviour units (BUTs) which communicate via a synchronisation kernel. An algorithm has been defined in Ref. 14 that corresponds to a set of rules for the evolution of BUTs and generation of new BUTs, in order to mimic the behaviour of the original Lotos specification. Abstract data type (ADT) specifications are in principle implemented as rewrite rules.

TOPO allows the use of annotations in specifications in order to represent aspects which cannot be formally represented in Lotos, such as time delays. More efficient ADT implementations hand-coded in C, default values or random choice for value negotiations events, mapping of events on occurrences of concrete interfaces, implementation side effects (e.g. print-out for implementation debugging) and busy waiting for events can also be incorporated in the final implementation by using annotations.

COLOS5 approaches the implementation problem in a slightly different way than TOPO. COLOS is targeted to the implementation of the behaviour part of annotated Lotos specifications which are rather near to the implementation, i.e. which contain already the implementation decisions which can be represented in Lotos. This means that the restrictions on the Lotos specifications that can be implemented with COLOS are more severe than in TOPO. For example, in contrast to TOPO, COLOS does not support the direct implementation of internal events. Furthermore, COLOS assumes that the ADT implementations are produced outside COLOS, imported as C code, and integrated during the implementation.

COLOS transforms a Lotos specification into a set of extended finite-state automata that cooperate using a synchronisation kernel. The synchronisation kernel handles synchronisation requests that are deposited in ports, while sorting out the possible events and scheduling pseudo-concurrency of the automata. Time-outs must be explicitly represented as events with timer components, and not as annotations such as in TOPO.

3. SOME ALGORITHMS FOR PROTOCOL DESIGN

This section presents three algorithms that have been tailored to protocol design. Section 3.1 presents a modification of the bi-partition of functionality algorithm of Ref. 13. 3.2 presents the regrouping of parallel processes of Ref. 4 and 3.3 presents a modification of the tableau method of Parrow.17

3.1 Bi-partition of functionality

Structuring of functionality can be performed in Lotos by using a transformation called functionality decomposition. A special case of functionality decomposition in which the LOTOS processes are replaced by two cooperating processes. The algorithm assumes that the process best partitioned is defined in Basic LOTOS, i.e. without the use of structured events and data types. Our modification supports some limited forms of data values.

We assume that the input specifications contains two gates and that it is represented in LOTOS by B[a, b]. We also assume that there is no internal event in the input specification and that there is no choice between event offers in which both gates a and b appear. In order to handle various different sorts of data values, we assume that the input specification and its derivatives have the following form:

\[ B[a, b] = \sum_k [a?d:Data_k \rightarrow req; b!d \rightarrow ind; B_k | k \in K] \text{ or } \sum_k [b?d:Data_k \rightarrow req; a!d \rightarrow ind; B_k | k \in K] \]

This means that for some finite index set \( K = \{1, \ldots, n\} \), each \( Data_k \) represents a distinct data sort and each \( B_k \) is a process of the same form as \( B \). Parameters \( req \) and \( ind \) represent that the direction of the data flow is from the environment to the process being decomposed and vice versa, respectively. In case \( K = \emptyset \), then \( B[a, b] = \text{stop} \).

Bi-partition will be performed in the gate set, such that
there will be one new process per gate. A bi-directional reliable medium capable of transferring one message or acknowledgement per direction of communication at a time can be represented by:

\[ M[p, q] = M_1[p, q] || M_2[p, q] \]

where

\[ M_1[p, q] = \sum_{p?m: Message_{req}; q!m!ind; M_1[p, q]} \]

This means that we want to build processes \( T_1[a, p] \) and \( T_2[b, q] \), such that the structure of the output specification will be:

\[ P[a, b] = \text{hide} \ p, \ q \text{ in } T_1[a, p] \| T_2[b, q] \| [p, q] \| M[p, q] \]

We also assume that after data has been "used" it can be deleted. Data is considered to be 'used' when it is forwarded to another component. Under these conditions we can define the algorithm for building \( T_1[a, p] \) and \( T_2[b, q] \) as:

- in case \( B[a, b] = \Sigma_e (a!d: Data_e!req; b!d!ind; B_e[k \in K]) \) then
  \[ T_1(B) = \Sigma_e (a!d: Data_e!req; p!Message_e(d)!req; p!Ack_e(Message_e(d))!ind; T_1(B_e)[k \in K]) \]
  \[ T_2(B) = \Sigma_e (q!m: Message_e!ind; b!Use_e(m)!ind; q!Ack_e(m)!req; T_2(B_e)[k \in K]) \]

- in case \( B[a, b] = \Sigma_e (b!d: Data_e!req; a!d!ind; B_e[k \in K]) \) then
  \[ T_1(B) = \Sigma_e (p!m: Message_e!ind; a!Use_e(m)!ind; p!Ack_e(m)!req; T_1(B_e)[k \in K]) \]
  \[ T_2(B) = \Sigma_e (b!d: Data_e!req; q!Message_e(d)!req; q!Ack_e(Message_e(d))!ind; T_2(B_e)[k \in K]) \]

The following signature defines the relationships between data and the corresponding messages exchanged through the medium:

\[ \text{Message}_e: \text{Data}_e \rightarrow \text{Message}_e \]
\[ \text{Use}_e: \text{Message}_e \rightarrow \text{Data}_e \]
\[ \text{Ack}_e: \text{Message}_e \rightarrow \text{Ack}_e \]

The condition \( \text{Use}_e(Message_e(d)) = d \), for all \( d \) of sort \( \text{Data}_e \), follows from the form of \( B[a, b] \).

The algorithm is proved correct if \( B[a, b] \approx P[a, b] \), i.e. the output specification is observation equivalent to the input specification. The proof consists of showing that \( \langle B[a, b], P[a, b] \rangle \) together with the identity relation define a weak bi-simulation relation.

### 3.2 Regrouping of parallel processes

During the design process it may occur that processes have to be regrouped, e.g. as an intermediate step in functionality decomposition. The LOTOS transformation that supports this is called regrouping of parallel processes. This transformation consists of regrouping processes composed in parallel in such a way that a given composition structure is matched. The behaviour of the resulting specification must remain the same as the input specification, therefore the input and output specifications must be strong bi-simulation equivalent in this case.

A formalisation of this transformation and a solution has been proposed in Ref. 5 and has been implemented in LITE. Particularly useful instances of this transformation (i.e. with fixed composition patterns) have been proved and applied in Refs 19 and 22.

We present here a restricted solution to this transformation, which is reported in Ref. 4. This solution makes use of the so-called Maximal Coordination Condition (MCC) in relation to the input and output specifications. A specification satisfies this condition if and only if every occurrence of the parallel operator in the specification has a set of synchronisation gates which is exactly the intersection of the set of observable gates of the argument behaviour expressions of the operator.

The algorithm requires that the input specification satisfies the MMC. The output specification is then simply derived from the given composition pattern by instantiating the parallel operators in the pattern in the unique way in which the MCC is satisfied.

For example, consider the following specification:

\[ B[a, b] = \text{hide} \ p, \ q \text{ in } (L_1[a] || L_2[b]) \| [a, b] \| (E_1[a, p] || p) \| E_2[b, p]) \]

which satisfies the MCC, and a desired composition pattern \( (L_1, \{1\}) \| (E_1, \{1\}) \| (L_2, \{2\}) \| E_2 \).

The input specification can be transformed according to the regrouping of parallel processes algorithm into:

\[ B'[a, b] = \text{hide} \ p \text{ in } (L_1[a] \| [a] \| E_1[a, p] \| [p] \| L_2[b] \| [b] \| E_2[b, p]) \]

The algorithm then guarantees that \( B[a, b] \sim B'[a, b] \).

### 3.3 Component construction

It would be desirable to have algorithms that accomplish functionality decomposition in a more general way than indicated in Section 3.1, i.e. even when a certain predefined structure of processes cannot be identified. Interactive algorithms can be defined in this case, in order to allow designers to fill in behaviours as a consequence of design decisions, such that by the end of this process a correct design is obtained.

The component construction algorithm can take as input any equation with unknown components to be designed, and supports the derivation of correct results in case these are possible. We illustrate the algorithm with the following equation form:

\[ B[a, b] \approx \text{hide} \ p, q \text{ in } ([\alpha[a, p]] || [\beta[b, q]] \| [p, q] \| M[p, q]) \]

where \( M[p, q] \) is a given specification of e.g. a communication service, and \( \alpha[a, p] \) and \( \beta[b, q] \) are the unknown components to be designed. The designer builds the unknown components of the output specification interactively, while at the same time (partial) verification is performed. A similar equation-solving procedure was proposed in Ref. 16 for CCS; our algorithm is a modification of this procedure for LOTOS.

The algorithm is based on transformations on a 'tableau' structure. A tableau consists of a goal and an environment. The goal indicates which equations have to be proved true, and the environment records the intermediate results of the unknown components. The environment is formally defined as a mathematical relation between identifiers and behaviours. Identifiers that are not assigned to behaviours (do not belong to the domain of the environment) are called free identifiers.

The following operations on tableaux can be performed.

---

THE COMPUTER JOURNAL, VOL. 35, NO. 5, 1992 483

31-2
4. AN EXAMPLE OF PROTOCOL DESIGN

This section illustrates a transformational approach to protocol design, using the algorithms presented in Section 3, with the design of a relatively simple, but non-trivial, protocol: a protocol that provides a question–answer service on top of an unreliable data transfer service. We also illustrate the application of the layering strategy: two successive decompositions are presented, resulting in two protocol layers. Each decomposition step is concerned with a distinct design problem, the nature of which determines what design methods are best suited for the design of the protocol during this step.

The first decomposition step deals with application concerns, i.e., it focuses on the provision of the application-oriented question–answer service (QAS) and assumes the availability of a reliable data transfer service (RTS). The procedure for the exchange of protocol information is very simple in this case, although the formatting and coding of the protocol information is potentially quite complex. Because of this, we use the bi-partition of functionality algorithm introduced in Section 3.1 in order to design the protocol, abstracting away from the structuring and representation of protocol information. The protocol designed during this step is called the question–answer protocol (QAP).

The second decomposition step deals with data transfer concerns, i.e., it aims at the provision of the reliable data transfer service assumed in the previous step on top of the unreliable data transfer service (UTS). Since this step takes into account possible loss of data, the procedure for the exchange of protocol information is relatively complex when compared to the first step. Due to this, we make use of the component construction algorithm introduced in Section 3.3 in order to design the protocol. A simple structuring of protocol information is considered, but not the representation of this information. The protocol designed during this step is called the reliable transfer protocol (RTP). Fig. 6 depicts these two decomposition steps.

![Figure 6. Protocol design example: (a) design objective; (b) design approach.](https://academic.oup.com/comjnl/article/35/5/478/402415)

This section is further structured as follows: Section 4.1 presents the architectural semantics of the concepts used in this example, 4.2 presents the specification of QAS, RTS and UTS, and the two design steps are subsequently elaborated in 4.3 and 4.4.

4.1 Architectural semantics

Two architectural concepts, namely service primitive and protocol data unit, and their representations in LOTOS are discussed below.
4.1.1 Service primitive

Services are defined in terms of service primitives. A service primitive is an elementary interaction between a service user (user for short) and the service provider (provider for short). We consider a simplified model of service primitives, which is suitable for our example.

- Service primitives are formally represented by (structured) events. The conditions imposed by a user or the provider to participate in a service primitive are represented by an event offer.
- Service primitives have a location, which is called a service access point. A service access point is represented by a gate identifier.
- Service primitives effect the exchange of information. We call the information exchanged in a service primitive a service information unit (SIU). An SIU exchanged in a service primitive is represented by an event parameter.
- Service primitives have a direction associated with the exchange of their SIU. A direction is represented by a second event parameter which can have one of two values: req (request), if the direction is from a user to the provider, and ind (indication), if the direction is from the provider to a user.

4.1.2 Protocol data unit

Protocol entities communicate through the exchange of protocol data units (PDUs). The exchange of PDUs is necessary to support the exchange of SIUs. Therefore PDUs convey (parts of) SIUs, and/or internally generated protocol control information that guarantees error-free exchange of SIUs.

Since there is no direct exchange of PDUs between protocol entities in our example, a PDU exchange is not represented by a single event. Instead, PDUs must be mapped on to data of service primitives of the underlying service in order to be exchanged. Mapping functions to relate PDUs to service primitives of the required service, and to service primitives of the underlying service, have to be defined.

4.2 Specification of the services

This section presents the simplified specifications of the services which are used in the protocol design example.

4.2.1 Question-answer service (QAS)

The question-answer service accepts a question from a calling user, delivers it to a called user, accepts an answer from the called user, and returns the answer to the calling user. This service bears some resemblance to the Remote Operations Service Element standardised by ISO, which is used in many distributed applications.

An instance of QAS with two users, each user with a fixed role (either calling or called), can be specified as follows, using a monolithic style:

\[
\text{QAS}[a, b] = \text{a}?: \text{q; Question}\text{! req; b! q? ind; b}?: \text{Answer}\text{! ind; stop}
\]

The specification of this service can alternatively be structured in terms of local and remote constraints, i.e. using a constraint-oriented style:

\[
\text{QAS}[a, b] = (L[a] \implies L[b]) \land (R[a, b] \implies R[a, b])
\]

4.2.2 Reliable data transfer service (RTS)

The reliable data transfer service accepts data from a sending user and delivers it to a receiving user. This means that the correct delivery is guaranteed. Communication networks which perform error detection and error correction provide this type of service.

For the specification of RTS we again assume two users, but in this case the users do not have a fixed role, i.e. either user may be a sending or a receiving user. Another restriction we impose is that the transfer capacity of the service for each direction of transfer is one data unit, i.e. at most one data unit can be 'under way' in a certain direction.

A specification which represents the two directions of transfer as independent constraints is:

\[
\text{RTS}[a, b] = RR[a, b] \land RR[a, b]
\]

4.2.3 Unreliable data transfer service (UTS)

The unreliable data transfer service accepts data from a sending user and attempts to deliver it to a receiving user. The service is unreliable in the sense that data is either delivered unchanged (not corrupted) to the intended user or it is lost (discarded). Communication networks which perform error detection but no (or not sufficient) error correction provide this type of service. We adopt the same restrictions for UTS as for RTS, i.e. the service is provided to two users and the transfer capacity per direction of transfer is one data unit.

Possible loss of data must be represented in the UTS specification. Whether data is lost or not depends exclusively on internal behaviour of the service provider, and therefore cannot be influenced by the users. The choice between delivery and loss is modelled by internal events (instances of i) in the following specification:

\[
\text{UTS}[a, b] = UR[a, b] \lor UR[a, b]
\]
Calling User  
\[ a \ ?q: \text{Question} \ \req \]

\[ a \ \la \ \text{lind} \]

\[ \text{QAS} \]

\[ \text{Called User} \]
\[ b \ ?q: \text{lind} \]

\[ \text{Figure 7. Behaviour of QAS.} \]

\[ \text{User A} \]
\[ a \ ?d: \text{Data} \ \req \]

\[ \text{RTS} \]

\[ \text{User B} \]
\[ b \ \la \ \text{lind} \]

\[ \text{RTS} \]

\[ \text{Figure 8. Possible primitive sequences of RTS.} \]

\[ \text{User A} \]
\[ a \ ?d: \text{Data} \ \req \]

\[ \text{UTS} \]

\[ \text{User B} \]
\[ b \ ?d: \text{Data} \ \req \]

\[ \text{UTS} \]

\[ \text{User A} \]
\[ a \ \la \ \text{lind} \]

\[ \text{UTS} \]

\[ \text{User B} \]
\[ b \ ?d: \text{Data} \ \req \]

\[ \text{UTS} \]

\[ \text{Figure 9. Possible primitive sequences of UTS.} \]

where

\[ UR_1[a, b] = a \ ?d: \text{Data} \ \req; (\{\text{delivery}\}; b \ ?d: \text{lind}; UR_1[a, b]) \]

\[ UR_2[a, b] = UR_1[b, a] \]

Figure 9 depicts some possible primitive sequences of this service.

4.3 Step 1: design of the question-answer protocol

Bi-partitioning the remote constraints which are defined by \( R_x \) and \( R_2 \) in (1) using the algorithm described in Section 3.1 yields:

\[ R_1[a, b] \approx \text{hide } p, q \text{ in } (R_{11}[a, p] \parallel R_{12}[b, q]) \parallel [p, q] \parallel M_1[p, q] \]

(4)

where

\[ R_{11}[a, p] = a \ ?q: \text{Question} \ \req; p \ !\text{Message}(q) \ !\req; p \ !\text{Ack}(\text{Message}(q)) \ !\text{lind}; \text{stop} \]

\[ R_{12}[b, q] = q \ ?p: \text{Message} \ !\text{lind}; b \ !\text{Use}(p) \ !\text{lind}; q \ !\text{Ack}(p) \ !\text{req}; \text{stop} \]

and

\[ R_2[a, b] \approx \text{hide } p', q' \text{ in } (R_{21}[a, p'] \parallel R_{22}[b, q']) \parallel [p', q'] \parallel M_2[p', q'] \]

(5)

where

\[ R_{21}[a, p'] = p' \ ?p: \text{Message} \ !\text{lind}; a \ !\text{Use}(p) \ !\text{lind}; p' \ !\text{Ack}(p) \ !\text{req}; \text{stop} \]

\[ R_{22}[b, q'] = b \ ?a: \text{Answer} \ !\text{req}; q' \ !\text{Message} \ !\text{lind}; q' \ !\text{Ack}(\text{Message}(a)) \ !\text{lind}; \text{stop} \]

Suffixes \( Q \) and \( A \) are used above instead of indexes from an index set \( K \); processes \( M_1 \) and \( M_2 \) represent different instances of a reliable medium (see Section 3.1). \( M_1 \) has gate set \{\( p, q \}\) and handles communication of type \( Q \), while \( M_2 \) has gate set \{\( p', q' \}\) and handles communication of type \( A \).

Applying regrouping of parallel processes in (4) and (5), and integrating \( M_1 \) and \( M_2 \) in a single communication medium \( M \), we transform (1) into the following protocol specification:

\[ QAP[a, b] = \text{hide } p, q \text{ in } ((L_1[a] \parallel R_{11}[a, p] \parallel R_{12}[a, p]) \parallel (L_2[b] \parallel [b] \parallel (R_{21}[b, q] \parallel R_{22}[b, q])))(p, q) \parallel M[p, q] \]

(6)

where \( M \) is defined as in Section 3.1 such that it handles communication of both types \( Q \) and \( A \).

Process \( M \) represents a dedicated channel for the exchange of PDUs. It is most cost-effective however if the protocol could use a general-purpose data transfer service, such as RTS defined in (2), since then the same service can be shared by many application protocols. In order to achieve this, we apply the operations \( \text{Wrap}, \text{UnwrapM}^k \) and \( \text{UnwrapA}^k \), with \( k \in \{Q, A\} \), and modify the protocol specification of (6) as follows:

\[ QAP[a, b] = \text{hide } p, q \text{ in } (QAP_1[a, p] \parallel QAP_2[b, q]) \parallel \text{RTS}[p, q] \]

(7)

where

\[ QAP_1[a, p] = L_1[a] \parallel (R_{11}[a, p] \parallel R_{12}[a, p]) \]

\[ R_{11}[a, p] = a \ ?q: \text{Question} \ \req; p \ !\text{Wrap}(\text{Message}(q)) \ !\req; p \ !\text{Wrap}(\text{Ack}(\text{Message}(q))) \ !\text{lind}; \text{stop} \]

\[ R_{12}[a, p] = p \ ?d: \text{Data} \ !\text{lind}; a \ !\text{Use}(\text{UnwrapA}(d)) \ !\text{lind}; p \ !\text{Wrap}(\text{Ack}(\text{UnwrapMA}(d))) \ !\text{req}; \text{stop} \]

\( L_1 \) as defined in (1)
4.4.1 Design decisions

Since there are a number of design decisions which have to be taken by the designer in order to reach a correct result. Therefore, some design decisions to be applied together with the component construction algorithm are considered here.

The provision of a reliable data transfer service implies that lost data should be re-sent by the sender until it is successfully delivered to the receiver. Hence the sender must be able to decide when it has to re-send data. We adopt the following solution.

- The receiver returns the sender an acknowledgement to indicate successful delivery of data.
- After it has (re-)sent data, the sender waits a time-out period for an acknowledgement. If the acknowledgement is received within this period, it assumes that the data is successfully delivered and stops (re-)seding data. Otherwise it assumes that the data is lost and sends it again.

Since an acknowledgement may also be lost by the underlying service, the sender may wrongly decide that data is lost. Consequently, the receiver may receive the same data more than once. The receiver must be able to decide whether data is duplicate or not. We adopt the following solution.

- An identifier (sequence number) is assigned to each item of sent data, such that successive data units have different identifiers. In this case it suffices to have two different identifier values, e.g. 0 and 1, which are alternately used.
- Duplicate data are acknowledged, but not delivered to the user of the required service.

In case the time-out period is too short, it may occur that the sender re-sends data while in fact neither the data, nor the acknowledgement is lost. Consequently, the sender may receive more than one acknowledgement related to the same data. The sender must be able to decide whether an acknowledgement is duplicate or not. We adopt the following solution.

- An identifier (sequence number) is assigned to each acknowledgement, such that this identifier corresponds to the identifier of the data is acknowledged.
- Duplicate acknowledgements are discarded.

4.4.2 Heuristics

The equations to be considered by the components construction algorithm when it is applied to (10) are always of the form:

\[ B[a, b] \equiv \text{hide} p, q in (X[a, p] \parallel Y[b, q]) \parallel M[p, q], \]

where each \( X \) and \( Y \) are unknown parts of the sender process and receiver process respectively, and \( M \) is a derivative of the unreliable data transfer service \( UTS \) in (10). Considering also the design decisions discussed above, we are able to identify some heuristics which can help in taking instantiation decisions when solving (10).

We assume that instantiations of \( X \) and \( Y \) have the form:

\[ X := \sum \{ x_i | i \in I \} \]
\[ Y := \sum \{ y_j | j \in J \} \]

where each \( x_i(y_j) \) is either:

- an initial event offer of \( B \), or
- an event offer which allows a synchronisation of \( X(Y) \)
and \( M \), such that the matching event of \( M \) is the first in a sequence of event offers which occurs at the common gate of \( X(Y) \) and \( M \).

An event offer that satisfies this condition is called a useful event offer. The designer must decide which subset of the set of useful event offers will be used in an instantiation.

We further assume that:
- if \( M \) has an initial event offer of type indication, the same event offer is only used in an instantiation of \( Y(X) \) if the information expected in this event offer can be derived from a preceding event of \( X(Y) \) and \( M \);
- an event offer of type request which allows a synchronisation of \( X(Y) \) and \( M \) is only used in an instantiation of \( X(Y) \) if the (wrapped) information exchanged in this event can be derived from a preceding event of \( X(Y) \) and \( M \), or from the parameters (stored status) of \( X(Y) \).

The latter heuristics can be used during instantiation to select properly from the set of useful event offers.

The following heuristics can be used to instantiate \( X(Y) \) by identification:
- \( X(Y) \) can be identified with a bound identifier \( V \), i.e. \( X = V(X = V) \), if the equations containing \( X(Y) \) are a subset of the equations containing \( V \), and \( X(Y) \) would be instantiated such that its initial events are the same as those of \( V \). In this case substituting \( X \) by \( V \) adds no new equations to the goal.
- \( X \) and \( Y \) can be simultaneously identified with the bound identifier \( V \) and \( W \) respectively, i.e. \( X = V \) and \( Y = W \), if the equations containing \( X \) and \( Y \) constitute a subset of the equations containing \( V \) and \( W \), and \( X \) and \( Y \) would be instantiated such that its initial events are the same as those of \( V \) and \( W \), respectively.

4.4.3 Application of the component construction algorithm

We start the application of the component construction algorithm with equation (10) to be solved and an empty environment. Table 2 lists the initial results of the algorithm, while indicating the operations which were performed to obtain these results. The initial value of the sequence number is taken to be 0.

For the sake of readability of Table 2, some additional process identifiers have been introduced. These are:

- \( RR_{[a,b]} = a ? d: Data! req; RR_{[a,b]}(d) \)
- \( UTS[p,q] = UR_{[p,q]} \uplus UR_{[p,q]} \)
- \( UR_{[p,q]}(d) = i; UR_{[p,q]}(d) = q! d! ind; UR_{[p,q]} \)
- \( Wrap: MessageD \rightarrow AckD \)
- \( UnwrapMD: Data \rightarrow MessageD \)
- \( Next: Number \rightarrow Number (* generate next sequence number *) \)

The following signatures:

- \( MessageD: Data, Number \rightarrow MessageD \)
- \( UseD: MessageD \rightarrow Data \)
- \( AckD: Number \rightarrow AckD \)
- \( Number: MessageD \rightarrow Number (* extract sequence number parameter *) \)
- \( Wrap: MessageD \rightarrow AckD \)
- \( UnwrapMD: Data \rightarrow MessageD \)
- \( UnwrapAD: Data \rightarrow AckD \)
- \( Next: Number \rightarrow Number (* generate next sequence number *) \)
Table 2. Application of the component construction algorithm

<table>
<thead>
<tr>
<th>Operation</th>
<th>Goal (equations)</th>
<th>Environment (instantiations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instantiation</td>
<td>$RR_1 \approx \text{hide } p, q \text{ in } (SE_1 \</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[p]?d: Data! ind[Number(NM(d)) = 1]; SE_{12}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$RE_1 = q?d: Data! ind[Number(NM(d)) = 0]; RE_{21}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[q]?d: Data! ind[Number(NM(d)) = 1]; RE_{22}$</td>
</tr>
<tr>
<td>Splitting Instantiation</td>
<td>$RR_{11} \approx \text{hide } p, q \text{ in } (SE_{11} \</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[p]?d: Data! ind[Number(NM(d)) = 1]; SE_{14}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$RE_{13} = q?d: Data! ind[Number(NM(d)) = 0]; RE_{15}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[q]?d: Data! ind[Number(NM(d)) = 1]; SE_{16}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$p!M(d, 0)! req; SE_{17}$</td>
</tr>
<tr>
<td>Splitting Instantiation</td>
<td>$RR_{11} \approx \text{hide } p, q \text{ in } (SE_{13} \</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[q]?d: Data! ind[Number(UnwrapMD(d)) + n];$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$RE_{23}<a href="n">b, q</a>$</td>
</tr>
</tbody>
</table>

Equations involving terms generated by these signatures are omitted for the sake of brevity.

A number of shorthand notations are used in Table 2 for representing concatenation of operations. These shorthand notations are:

$NA = \text{Number } \ast \text{UnwrapAD}$

$NM = \text{Number } \ast \text{UnwrapMD}$

$M = \text{Wrap } \ast \text{MessageD}$

$D = \text{UseD } \ast \text{UnwrapMD}$

Furthermore, process instantiations are abbreviated in the table by omitting gates and parameters (e.g. we use $UR_{11}$ for $UR_{11}[[p, q](d)]$).

Repeated application of the operations of the component construction algorithm eventually leads to a tableau without free variables in the equations of the goal. A solution for the sender and receiver process can then be constructed. In order to ensure that this solution is a correct one, the remaining equations in the goal should be proved true. This can be done by repeatedly applying splitting and removal operations until all equations are removed in the goal of the final tableau.

The solutions found for $SE_{1}$ and $RE_{2}$ are then:

$SE_{1}[a, p](n: \text{Number}) = a?d: Data! req; SE_{1}[a, p](n, d)$

$[p]?d: Data! ind[Number(UnwrapAD(d)) + n]; SE_{1}[a, p](n)$

where

$SE_{1}[a, p](n: \text{Number, } d: Data) = p! Wrap(MessageD(d, n)) req; SE_{1}[a, p](n, d)$

$[p]?d: Data! ind[Number(UnwrapAD(d)) + n]; SE_{1}[a, p](n, d)$

$RE_{1}[a, p](n: \text{Number, } d: Data) = p?d: Data! ind [Number(UnwrapAD(d)) = n]; SE_{1}[a, p](Next(n))$

$[p]?d: Data! ind[Number(UnwrapAD(d)) + n]; SE_{13}[a, p](n, d)$

$[p]! Wrap(MessageD(d, n)) req; SE_{13}[a, p](n, d)$

$RE_{1}[b, q](n: \text{Number}) = q?d: Data! ind [Number(UnwrapMD(d)) = n]; RE_{23}[b, q](n, d)$

$[q]?d: Data! ind[Number(UnwrapMD(d)) + n]; RE_{23}[b, q](n)$

where

$RE_{23}[b, q](n: \text{Number}) = q! Wrap(AckD(Next(n))) \ | req; RE_{23}[b, q](n)$

$RE_{23}[b, q](n: \text{Number, } d: Data) = b! UseD(UnwrapMD(d)) \ | ind; RE_{23}[b, q](n)$

$RE_{23}[b, q](n: \text{Number}) = q! Wrap(AckD(n)) \ | req; RE_{23}[b, q](Next(n))$

$[q]?d: Data! ind[Number(UnwrapMD(d)) = n]; RE_{23}[b, q](n)$

Fig. 10 depicts an alternative representation of the sender and receiver process by way of state transition diagrams, using the shorthand notations of Table 2.

Finally, it follows from (9) that the protocol entities are:

$RTPE_{a, p} = SE_{a, p}(0) \ ||| RE_{a, p}(0)$

$RTPE_{b, q} = SE_{b, q}(0) \ ||| RE_{b, q}(0)$

where

$RE_{a, p}(n: \text{Number}) = RE_{a, p}(n)$

$SE_{a, p}(n: \text{Number}) = SE_{a, p}(n)$

Hence, the protocol is defined by:

$RTP[a, b] = hide p, q \text{ in } (RTPE_{a, p} \ ||| RTPE_{b, q}) \ | [p, q] \ | UT[p, q]$
implementation of protocols. Special attention is given to a transformational approach towards protocol design, illustrated by means of a simplified protocol design example.

Protocol designers are interested in design methods that altogether cover the complete design trajectory. Considering the design methods presented and discussed in this paper, we can make the following observations.

(1) Specifications that are structured according to specification styles facilitate validation. The use of a few related specification styles, as proposed in Ref. 21, contributes to the structuring of specifications along the architectural and implementation phase of the protocol design trajectory.

(2) The transformation approach illustrated in Section 4.3 can be used in the early phases of protocol design, when the distribution of basic protocol functions that directly support the required service is considered. This transformation is particularly suitable if the required service is application-oriented rather than data transfer-oriented. The reason for this is that application-oriented services are generally defined in terms of different service elements, involving different service primitives. For each service element, separate protocol functions, involving different PDUs, can be defined that require a reliable data transfer service for the exchange of PDUs. Therefore the decomposition of functionality in application and data transfer becomes straightforward.

(3) The transformational approach illustrated in Section 4.4 can be used in subsequent steps of protocol design, where protocol functions are defined that are more concerned with the hiding of undesirable properties of the underlying service, rather than merely providing properties of the required service.

(4) It may happen that the approach illustrated in Section 4.4 is not applicable for some instances of design, since the instantation of free variables may become unmanageable due to lack of appropriate heuristics. In this case the refined protocol has to be designed by hand and validated using testing or verification afterwards.

(5) The protocol implementation can be structured in terms of resources that correspond to PDICs. The PDICs approach mentioned in Section 2.4.1 can be used to derive an implementation of the protocol in a particular target environment. This covers the last part of the implementation phase of the design trajectory. An alternative to the use of PDICs is to refine the protocol specification until a specification is reached which can be mapped straightforwardly on to a realised. An example of the latter approach is given in Ref. 10, where a resource-oriented specification is first transformed into a state-oriented one and subsequently mapped on to C code.

(6) Another approach towards protocol implementation is the use of compilers. The more realistic and effective approach, however, is to combine the use of compilers with the use of PDICs. COLOS seems to be the compiler which is most suitable for the hybrid approach.

(7) Since during the protocol design and implementation processes ad hoc methods may be used, designers have to validate the protocol implementations against their specifications. Furthermore, protocol implementations very often have to be certified, which means that organisations with a certain authority must state that the implementation complies to certain standard specifications. Conformance testing methods can be used for both validation and certification of protocol implementations. Conformance testing methods using LOTOS have been discussed in Ref. 2.

Although a sophisticated set of design methods and support tools have been developed in the Lotosphere project, some effort is still necessary to integrate methods and to enhance tool functionality. It is our firm belief that the Lotosphere design methodology and its tools constitute an important step forward in the direction of efficient and correct protocol design and implementation.

REFERENCES

13. R. Langerak, Decomposition of functionality: a correctness-preserving LOTOS transformation. In *Protocol Speci-

Book Review

ROGER BAILEY. Functional Programming with Hope. Prentice-Hall, Hemel Hempstead. £33.50. ISBN 0 13 338237 0

Functional programming is claimed to offer two kinds of promise. (1) Because functional programs obey mathematical laws, they can be systematically derived or proved correct. This could ease the software crisis. (2) Because the operands of a pure expression can be evaluated in any order, functional programs can be executed on parallel hardware.

In spite of some notable advances, the software crisis is still with us, and nobody associates functional programming with high-performance computing. But functional programming has one undoubted success: in education. Increasing numbers of students learn functional programming in their very first programming course. A functional language reveals the connections between programming and mathematics, and breaks bad habits acquired from BASIC.

A difficulty with teaching functional programming first is that good textbooks are scarce. The pace is usually wrong. Bird and Wadler's Introduction to Functional Programming is valuable, but it is closer to a research monograph than a beginner's text. Abelson and Sussman's classic Structure and Interpretation of Computer Programs presents many remarkable examples of functional programming but, again, it is too difficult for beginners. Many other books (including one of mine) do not even pretend to be suitable for beginners. Wikström's Functional Programming Using ML has the opposite problem: students find it too slow.

Roger Bailey's Functional Programming with Hope has the pace just about right. Chapter 1 discusses the basic types (integers, characters and so forth) and the evaluation of expressions. Chapter 2 introduces functions and variable binding. Chapter 3 introduces lists and pattern matching. Chapter 4 presents a great many examples involving data type definitions and trees. Chapter 5 discusses Hope's polymorphic type system. The next two chapters concern higher-order functions, which operate upon other functions as arguments and results. Chapter 8 discusses lazy lists, while Chapter 9 applies them to input and output. The appendices include answers to the exercises and syntax charts for Hope. There is a rather sparse index.

A beginner's text should not be too fast; this one is perhaps too slow. In places there are too many examples, and it is hard to see where they are leading. The higher-order function map, which applies a given function to every element of a list, appears in four variations, and is illustrated in connection with seemingly every other feature of Hope. I would have liked to see some large examples treated at length, and more discussion of programming techniques. For example, list reversal appears only as an exercise; the function given as the solution requires quadratic rather than linear time.

Which language should an introductory course adopt? Many courses use ML; Hope is a closely related functional language. Both are notable for their polymorphic type systems. Polymorphic type checking prevents run-time type errors, while admitting functions that have general types. The list reversal function works for all lists, regardless of their element type, and the type checker notes that the function's result has the same type as the argument. Both languages allow functions to be written in the form of pattern transformations.

I am experienced with ML, and prefer it to Hope for teaching. ML deduces the types of functions automatically, while Hope requires type declarations for all functions. Although type declarations make programs easier to read, they are a great deterrent when students work interactively. The type of a polymorphic function can be difficult to deduce, especially for beginners. Hope also contains many syntactic hazards, such as the where declaration, whose scope extends in both directions. Bailey devotes much space to advice on correct bracketing.

Hope is a purely functional language. A Hope expression has no observable effect on the program state. Input from the terminal takes place via a lazy list of characters; characters join the back of the list as they are typed, and leave the front of the list as the program demands them. Output is treated similarly. This treatment of input/output is purely functional, but involves many complications. ML skirts the issue by providing imperative input/output facilities. Thus, ML also supports conventional programming, while Hope is suitable for teaching functional programming in its pure form.

When we consider that most students learn programming through Fortran, BASIC, C or even assembly language, the differences between Hope and ML fade into insignificance. In both languages, programs are exceptionally clear and concise; students can express sophisticated algorithms in a few lines of code.

Functional Programming with Hope is one of the best textbooks available for an introductory programming course using a functional language. The only serious problem is the price: at £33.50, it is out of reach of many students. We can only hope that the publishers bring out an inexpensive version soon.

L. C. PAULSON
Cambridge