Specification styles in distributed systems design and verification

Chris A. Vissers, Giuseppe Scollo, Marten van Sinderen and Ed Brinksma

University of Twente, Fac. Informatics, 7500 AE Enschede, The Netherlands

Abstract

Vissers, C.A., G. Scollo, M. van Sinderen and E. Brinksma, Specification styles in distributed systems design and verification, Theoretical Computer Science 89 (1991) 179-206.

Substantial experience with the use of formal specification languages in the design of distributed systems has shown that finding appropriate structures for formal specifications presents a serious, and often underestimated problem. Its solutions are of great importance for ensuring the quality of the various designs that need to be developed at different levels of abstraction along the design trajectory of a system. This paper introduces four specification styles that allow us to structure formal specifications in different ways: the monolithic, the constraint-oriented, the state-oriented, and the resource-oriented style. These styles have been selected on the basis of their suitability to express design concerns by structuring specifications and their suitability to pursue qualitative design principles such as generality, orthogonality, and open-endedness. By giving a running example, a query-answer service, in the ISO specification language LOTOS, these styles are discussed in detail. The support of verification and correctness preserving transformation by these styles is shown by verifying designs, expressed in different styles, with respect to each other. This verification is based on equational laws for (weak) bisimulation equivalence.

1. Introduction

1.1. The role of system architecture

A design trajectory consists of a number of designs of the same system that differ in abstraction level, especially with respect to the way in which user requirements are reflected and the measure in which design decisions are incorporated. The first, most abstract, design of a system along the trajectory, often referred to as the "(system) architecture", should express only the user requirements for the realization of the system in the form of a product. This architecture is used as a point of reference for the derivation, via a number of intermediate designs, of a concrete design, often called "implementation". The implementation acts as a blueprint for the realization. It fulfils not only the user requirements, but in addition also incorporates a large number of design decisions. Such design decisions, or "implementation decisions", need to be taken in order to make a product but are irrelevant to the user requirements. Obviously, implementation decisions can be taken in many

different ways, thus allowing many valid product implementations of the same architecture.

The concept of architecture plays an important role in the design and implementation of open distributed systems. The objective of open distributed systems is to allow the correct interworking of components of distributed systems, each of which may be partly or completely produced by different and independent manufacturers. This objective can only be achieved by standardizing precise specifications of the rules of interworking. Important examples of such specifications are the ISO and CCITT protocol and service standards for Open Systems Interconnection [14]. An important requirement is that the standards are defined in an *implementation-independent* way. Manufacturers should have a maximal freedom to implement products according to their own insights, capabilities, and options. Thus, any element in a specification that constrains this freedom unnecessarily is in conflict with the goal of openness. This implies that such a specification has to be an abstract object defining user requirements only, i.e. as a system architecture.

An architectural specification has to observe at least two important (and often confused) abstraction criteria to fulfil its role of an implementation-independent definition.

(1) The architecture should be defined in terms of observable behaviour only, like a "black box". We will refer to this as an *extensional* definition. In principle, one should *not* define the architecture in terms of a structure that discloses details of an internal organization that may be given to implementations. The latter will be referred to as an *intensional* definition.

(2) The specification should reflect faithfully the architectural concepts at stake [35], such as e.g. protocol, service, interface, etc. This implies that the specification language adopted must support extensional definition, i.e. it should possess a syntax and semantics that enable one to express the aforementioned observable behaviour directly and naturally, instead of by clever but opaque encoding tricks. We use the term "architectural semantics" to denote the relationship between the primitive language constructs and their interpretation in terms of basic architectural concepts [19].

1.2. Pragmatics of architectural specification

Having emphasized the great importance of the above abstraction criteria, we observe that in practice they are very difficult to meet for the specification of any architecture of more than elementary complexity. In practice, such architectures need to be structured in order to keep them comprehensible, and to efficiently express their functionality. Structuring, however, may easily introduce implementation-oriented elements in the architecture.

The technical complexity of most architectures forces the specifier to structure them in terms of simpler functional entities, even if the preservation of extensionality is an explicit goal. To achieve well-structured specifications of an architecture one needs an adequate set of basic architectural concepts and construction rules that

allow one to define it as a composition of those basic concepts (cd. the OSI Reference Model [14]). Such internal structure usually mixes architectural considerations with implementation pragmatism.

The efficient expression of functionality is dictated by the use of general-purpose specification languages: it is impractical, if not unfeasible, to develop and use specification languages that contain primitive constructs for each potential architectural requirement. This implies that one needs an adequate set of generic language elements and construction rules, that allow one to faithfully express an architectural requirement as a composition of such language elements. This usually means that the same requirement can be expressed by many different compositions, and therefore here too the extensionality requirement risks violation.

From the above it follows that the introduction of structure and thus of intensional elements in a specification, is hardly avoidable in practice. Of course, an implementor, i.e. the designer of an implementation, could try to ignore such structure. He or she could try to derive the extensional behaviour from the given architectural specification and use this "extensional detour" as the starting point for developing an implementation. However, since the structure in the specification was introduced for comprehensibility and reasons of efficiency, this detour is likely to be (very) hard and cumbersome, and hence of doubtful practical value. In practice, therefore, the implementor will use the architecture *with* its given specification structure as the starting point for implementation. Hence it is likely that this structure will be reflected in the structure of the implementation.

We conclude that, although an architect should refrain in principle from intruding in the realm of implementation, he or she carries a considerable responsibility for the structure, and thus the quality, of the implementation in practice. The objective of this paper is to investigate how an architect can exploit, rather than ignore, this responsibility to pursue explicit design objectives with the goal of advancing the quality of implementations [25].

The approach we propose in this paper is based on the concept of *specification style*. We advocate the use of a few well-chosen styles that allow one to structure formal specifications and that can be used to pursue qualitative design principles. The use of common and mutually related specification styles is also important to preserve homogeneity of large specifications which are usually developed by (large) teams of specifiers. Such styles would enable the designer to control correctness throughout the design trajectory and thus to produce higher quality designs in a shorter time.

Four specification styles are introduced in this paper, viz. the monolithic, the contraint-oriented, the state-oriented, and the resource-oriented specification style. For each style it is globally indicated which design objectives can be supported. The styles can be used to support successive steps of the design trajectory from architecture to implementation. This is demonstrated by a verification exercise in the context of a simple example. The evaluation of the constraint-oriented style is addressed in more detail. This style, suggested in [34], defines an object in terms

of a set of constraints, each of which can be chosen so as to correspond closely to a natural requirement on the behaviour of the object. It appears to be very effective in initial design phases when requirements capturing is the prime objective.

This paper is further organized as follows. In Section 2 we relate specification styles to qualitative design criteria such as orthogonality, generality and open-endedness. In Section 3 we introduce a number of specification styles and illustrate them by means of examples. In Section 4 we illustrate how specification styles can be used to achieve correct designs. In Section 5 we evaluate the relative merits of the discussion styles, in particular the constraint-oriented style, and evaluate their roles and interworking in the system design trajectory. Section 6 contains our conclusions.

2. Specification style and design principles

The term style is usually associated with someone's particular approach to a certain field of expertise. This applies also to the development of specifications since the infinite number of possibilities to structure specifications invites individual specifiers to give a personal touch to their work. It would, however, be very disadvantageous for an implementor of standards to be confronted with a mixture of such personal touches as this would result in specifications with quite different expressions of the same or similar architectural concepts. Such redundancy is confusing and invites different implementation constructs, or conventions, where one could suffice.

This confusion can be avoided by adhering to a specification discipline where a limited number of effective specification styles are defined, and related to the design objectives that they support. Such an approach was first proposed in [36]. With the availability of commonly agreed styles, (groups of) specifiers can determine a strategy for applying them in an architectural design, and report this strategy explicitly in the informative, explanatory documentation which accompanies the formal text of their specifications.

Before considering particular styles, it seems appropriate to pay some attention to the relation between architectural design principles [2] and specification styles. General design principles as orthogonality, generality and open-endedness can be taken as the foundation of design methods for open distributed systems, as is illustrated in [37]. It appears that these principles are also useful to guide the specification of such systems [28].

Orthogonality. *Independent architectural requirements should be specified by independent definitions.*

The principle of orthogonality acts as a criterion to evaluate styles according to their suitability to recognize and emphasize functional independence. A style where

independent concerns are easily intermingled, for example if it does not allow the identification of locality aspects, is certainly less useful than a style that allows to separate them.

Generality. *Generic, parameterized definitions should be preferred over collections of special-purpose definitions.*

This principle induces the habit of recognizing and bringing out "the bare essence" of a problem. Whenever an essentially new design problem is encountered, a specifier should produce generic, reusable solutions to the advantage of future efforts. Consequently, a specifier should seek to reuse existing solutions. Dedicated, special purpose architectures can be obtained by instantiating general-purpose ones, tuning the design to the specific goals required by the particular application. This can only be supported in a specification formalism that allows for parameterized constructs.

Open-endedness. *Designs should be maintainable, i.e. easy to extend and modify.*

Architectures will usually be maintained. Not only to repair errors, but also to extend or modify their functionality to obtain better, e.g. more cost-effective, designs. With complex architectures it is often the case that the costs of such maintenance by far exceed the sum of all other design costs. Moreover, in the case of a new architecture having a reasonable amount of commonality with existing ones, it is advantageous to start the design by modifying existing, known solutions, rather than just beginning from scratch.

The application of open-endedness is more than a combined application of the principles of orthogonality and generality: it requires a modular, iterative decomposition of a design into small, rearrangeable units of specification. The availability of suitable composition constructs depends in the first place on the extent to which they are supported by the particular specification formalism. But given a formalism the modularity and rearrangeability of a specification depend on the style of the specifier.

3. Examples of specification styles

In this section we identify four specification styles and assess their relation to the design objectives and principles that characterize distinct phases of the system design trajectory. We also address the relationship between specification style and formal description in the specification language LOTOS [17, 3]. We illustrate this with simple examples in LOTOS. The interested reader may find more complex examples of these styles, based on the application of LOTOS for large scale specifications, in [29, 32, 20]. Some considerations concerning styles of specification of abstract data types in LOTOS can be found in [12].

Specification styles can be characterized as being supportive of either extensional or intentional description. This characterization relates the styles to their effectiveness for the different phases along the design trajectory from architecture to implementation.

Extensional description

In the first phase of the design of a system, when a (formal) definition of the architecture is the target, abstractness of the definition is the prime concern. For the effectiveness of a specification style in this phase it is essential that it supports *extensional* description. As was indicated in the first section an extensional description defines a system in terms of its external observable behaviour, viz. in terms of the interactions between the system and its environment. No internal boundaries or interactions between parts of the system shall be the subject of specification, nor any representation of the internal state space.

An extensional description is the proper starting point for system design because it deals with the definition of the functions of the total system: the "requirements capturing" or the *what* of the design. The external behaviour can then be considered in its own right by the system architects and by the system users exactly as it is presented in the extensional definition of the architecture.

Two extensional specification styles are identified. They are referred to as the monolithic (Section 3.1) and the constraint-oriented (Section 3.2) style.

Intensional description

In the second phase of the design of a system, the implementation phase, the internal organization, or the *how*, of the system is of concern. For the effectiveness of a specification style in this phase it is essential that it supports *intensional* description. An intensional description defines a structuring of the system in terms of interacting parts, or in terms of an explicit definition of the state space of (the parts of) the system, or both. The obvious objective is to relate the parts or state information to implementable constructs. We will use the term "resource" to denote the parts or entities out of which a system is constructed.

A description can be more or less intensional, depending on the depth of the substructuring it contains, and on the amount of detail that is present concerning the internal states and/or mechanisms embodied by (the resources of) the system. Two intensional specification styles are identified. They are referred to as the state-oriented (Section 3.3) and the resource-oriented (Section 3.4) style.

Example: informal specification

To help the reader understand the differences between the specification styles, and to show the transformation of a specification from one style into another; we have illustrated each style with a different specification of the same, simple system.

For this system we have chosen the "question/answer service" shown in Fig. 1. A question is generated by a user Q and forwarded by the service to another user
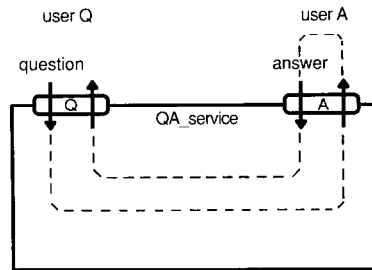
Fig. 1. Question/answer service.

A. The latter is required to generate an answer which is returned by the service to user Q. The question/answer service corresponds to the conventional request/response service [18].

The presentation of each style is organized as follows:

- characterization of the style,
- objectives of, or reasons for, using the style and the relationship with the design principles presented in Section 2,
- an example specification of LOTOS that corresponds to the informal specification above,
- the impact of the style on usage of language features, in particular specific LOTOS operators.

3.1. Monolithic style

Characterization

In the monolithic style only observable interactions are presented and ordered as a collection of alternative sequences of interactions in branching time.

Relation to design objectives and principles

This style forces the specifier to show the temporal relationship between observable interactions directly and prohibits the possibility of providing hints concerning implementation-oriented aspects, such as, for example, internal structure, or locality aspects of the observational behaviour. A monolithic description is therefore as implementation-independent as possible.

The monolithic description can be very useful if the system can be defined and understood as a simple black box (as is the case with our example). It can also provide a useful starting point for developing specifications in the other styles given below. In general, though, it is of little practical use for the design of more complex distributed systems because, through its lack of structure, it is not suitable for human comprehension of, and reasoning about, these systems. Consequently, it is more applicable to the specification of entities with a simpler functionality (e.g. service specifications) than to the more complicated ones (e.g. protocol specifications).

The principles of orthogonality and open-endedness are violated by this style, whilst generality can only be supported to a very limited extent.

Example 1 (*monolithic specification of a question/answer service*). The specification below is a straightforward representation of the question/answer service as given in the informal definition.

> **process** *QA_service [Q,A]*: **noexit** :=
> *Q?q:question; A!q; A?a:answer; Q!a;* **stop**
> **endproc**

The above example seems, in comparison to the representations given in the other styles, to be the simplest way to express this service. It appears, however, that the monolithic style will soon generate unreadable descriptions when the functionality of a system gets more complicated. As an exercise, the reader is invited to consider the specification, using the monolithic style, of a question/answer service that supports multiple concurrent pairs of users.

Relation to language constructs

The only constructs allowed are sequential composition, choice, guards, and tail-recursion. The prohibition of the use of parallel operators, in particular, deprives the specifier of his best means to achieve a separation of concerns and, in many cases, conciseness.

3.2. Constraint-oriented style

Characterization

In the constraint-oriented style only observable interactions are presented, but their temporal ordering is defined by a conjunction of different constraints.

Relation to design objectives and principles

This style supports extensional description with modularity and parallel composition. The structuring is based on the identification of the different (design) constraints that determine the external behaviour ("separation of extensional concerns"). However, by structuring the external behaviour, this style may in some cases provide useful hints for internal structuring.

An architectural criterion for the constraint-oriented specification of distributed systems is to separate local constraints from remote, or end-to-end, constraints. This approach is extremely useful as a preparation for the implementation trajectory, where different parts of the distributed system have to be allocated to different implementation authorities. It also allows one to reuse the constraints in a specification at a lower abstraction level where the internal structure is shown explicitly. Another useful criterion for the use of this style is to identify orthogonal functions by separate constraints.

All design principles mentioned in Section 2 can be supported by the constraint-oriented style.

Example 2 (*constraint-oriented specification of a question/answer service*).

process *QA_service [Q,A]:* **noexit** :=
   (QA_local [Q] ||| *QA_local [A])*
||
   QA_remote [Q,A]
where
   process *QA_local [X]:* **noexit** :=
   X ?x:question; X ?y:answer; **stop**
   endproc
   process *QA_remote [X, Y]:* **noexit** :=
   X ?x:question; Y!x; **stop** ||| *Y?y:answer; X!y;* **stop**
   endproc
endproc

This specification of *QA_service* formalizes a decomposition of the overall external behaviour into local (i.e. local at gate *Q* or local at gate *A*) and end-to-end (i.e. between gate *Q* and gate *A*) constraints. The local constraint at either side can be specified by proper (gate) instantiation of the same generic process, namely of *QA_local*, thus supporting the design principles of generality and conciseness.

Relation to language constructs

This style induces an extensive usage of parallel composition operators. The composition may be unsynchronized ("interleaved'" in LOTOS), or synchronized, depending on whether the separate constraints apply to disjoint or nondisjoint interaction sets, respectively. In addition, generic constraints call for extensive usage of parameterization. Because of the intended extensional character of this style the application of the hiding operator is prohibited.

3.3. State-oriented style

Characterization

With the state-oriented style the system is regarded as a single resource whose internal state space is explicitly defined. This style, therefore, presents only observable interactions—unstructured, except as a collection of alternative sequences of interactions—and a representation of the global state space that is manipulated by these interactions.

Relation to design objectives and principles

The use of the state-oriented style provides insight in the amount of state information to be maintained by a resource and the complexity of the manipulation of this

information. This insight can be used to transform the formal specification into a final implementation of the resource by using it to develop the data structures and programs that embody the machine executable code. The state-oriented style is therefore particularly useful in the final phase of the design trajectory when the resource can be defined and understood as an object that can be implemented by a single implementation authority.

The state-oriented style is of limited use, however, for the abstract specification of distributed systems. This has at least two reasons. First, the state-oriented style is analogous to the monolithic style in the presentation of dynamic behaviour. By its lack of structure it confronts the reader with the same problems of lack of surveyability and comprehensibility. Second, for distributed systems and for complex resources in general, one must decompose the state space. This implies the iden-tification of state variables with appropriate domains, which appears to be an extremely difficult problem. This problem is again aggravated by the "fine grain" of the state-oriented style, as the individual description of each state transition lacks structural hints.

With respect to the design principles, the same remarks apply as for the monolithic style.

Example 3 (*state-oriented specification of a question/answer service*). The specification below uses five data values to represent five global states. Alternatively, these global states could also have been written as five pairs of local states, each pair consisting of a local state associated with gate Q and a local state associated with gate A. This approach is followed in the specification of OSI services with state tables (e.g. in [15, 16]).

> **process** QA_service [Q,A]: **noexit** :=
> **choice** q:question, a:answer ‖ QA_Service1 [Q,A] (awaitQ, q, a)
> **where**
>   **process** QA_service1 [X,Y] (s:state, x:question, y:answer): **noexit** :=
>   [s = awaitQ] → X?x1:question;
>   QA_service1 [X,Y] (pendingQ, x1, y)
>   ‖ [s = pendingQ] → Y!x;
>   QA_service1 [X,Y] (awaitA, x, y)
>   ‖ [s = awaitA] → Y?y1:answer;
>   QA_service1 [X,Y] (pendingA, x, y1)
>   ‖ [s = pendingA] → X!y;
>   QA_Service1 [X, Y] (done, x, y)
>   ‖ [s = done] → **stop**
>   **endproc**
> **endproc**

Relation to language constructs

The same comments apply as for the monolithic style.

3.4. Resource-oriented style

Characterization

In the resource-oriented style both observable and internal interactions are presented. The behaviour in terms of the observable interactions is defined by a composition of separate resources in which the internal interactions are hidden. In turn, these resources may be specified using any style.

Relation to design objectives and principles

This style supports intensional description with modularity and parallel structures. The structure is based on the distinction between resources by the separation of "intensional concerns" in much the same way as constraints in the constraint-oriented style are derived by separating the extensional concerns. The development of an internal structure is guided by the following criteria:

• each resource should represent a self-contained entity with simple and efficient interfaces to other resources,
• each resource should be easier to implement than the original system and
• each resource should concern the smallest number of implementation authorities. Resource orientation "in extremo" identifies resources such that each one can be assigned to a single implementation authority and is specified in constructs with straightforward mappings onto, possibly predefined, implementation constructs.

Since the resources themselves can be specified using any style, including the ones introduced here, it is the resource-oriented style in particular that allows to apply styles iteratively, resulting in a style-supported design methodology. The resource-oriented style also allows for considerations of performance. First, (e.g. identical) resources can be identified which can act independently (*parallelism*) and second, resources can be identified which have to interact, but work on different aspects of a task (*cascading*).

All design principles presented in Section 2 are supported by the resource-oriented style.

Example 4 (*resource-oriented specification of the question/answer service based on a protocol*). The internal structure of the system can, in principle, be given in an infinite number of ways. For reasons of brevity and simplicity we show only one. This example assumes that the *QA_service* cannot be implemented directly. Therefore, local resources in the form of protocol entities are identified that can be implemented separately and that interwork via a lower level service to provide the *QA_service*. The resulting internal structure of the system is show in Fig. 2. For the lower level service an easier implement, connection-less data transport service is chosen. This service is not concerned with the question/answer relationship.
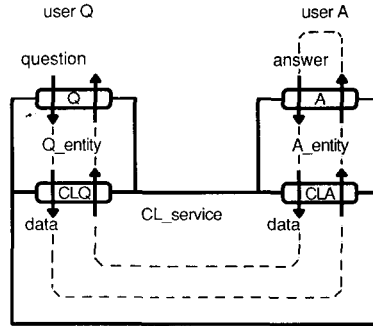
Fig. 2. Question/answer protocol.

process *QA_service [Q,A]:* **noexit** :=
hide *CLQ,CLA* **in**
    (*(Q_entity [Q,CLQ]* ‖‖ *A_entity [A, CLA])*
    |[CLQ,CLA]|
        CL_service [CLQ,CLA]
)
where
    (* process definitions *Q_entity*, *A_entity* and *CL_service* *)
endproc

To provide a complete specification each of the resources identified by the use of the resource-oriented style has to be specified. We elaborate the specifications of some of these resources using the monolithic, constraint-oriented and state-oriented style, respectively and indicate some relations with the corresponding specifications of the integral *QA_service*.

Example 4.1 (*monolithic specification of Q_entity, A_entity and CL_service*). The protocol entity *Q_entity* (*A_entity*) is able to send (receive) a question and to receive (send) an answer that is encoded as data via the lower level service. Note that the interactions specified in *QA_local* in Example 2 are interleaved with internal interactions to form the specification of the local protocol entities. Also note the rather tedious specification of *CL_service*, which is caused by the use of the monolithic style.

    process *Q_entity [X, Y]:* **noexit** :=
    X?x:question; Y!send!encode_q(x);
    Y!receive?d:data; X!decode_a(d); **stop**
    endproc

    process *A_entity [X, Y]:* **noexit** :=
    Y!receive?d:data; X!decode_q(d);
    X?y:answer; Y!send!encode_a(y); **stop**
    endproc

process *CL_service* [*X, Y*]: **noexit** :=
   X!*send*?*d*1:*data*;
   (*Y*!*receive*!*d*1; *Y*!*send*?*d*2:*data*; *X*!*receive*!*d*2; **stop**
   ‖ *Y*?*send*?*d*2:*data*;
      (*Y*!*receive*!*d*1; *X*!*receive*!*d*2; **stop**
      ‖ *X*!*receive*!*d*2; *Y*!*receive*!*d*1; **stop**
))
  ‖ *Y*!*send*?*d*2:*data*;
   (*X*!*receive*!*d*2; *X*!*send*?*d*1:*data*; *Y*?*receive*!*d*1; **stop**
   ‖ *X*!*send*?*d*1:*data*;
      (*X*!*receive*!*d*2; *Y*!*receive*!*d*1; **stop**
      ‖ *Y*!*receive*!*d*1; *X*!*receive*!*d*2; **stop**
))
  endproc

Example 4.2 (*constraint-oriented specification of Q_entity*). The transformation of a constraint-oriented specification into a resource-constraint-oriented specification (i.e. a resource-oriented specification where the resources are defined using the constraint-oriented style) allows constraints defined at the higher abstraction level to be reused at the lower abstraction level. In the example below the constraint-oriented specification of *Q_entity* reuses the process abstraction *QA_local* defined in Example 2. This approach can simplify considerably the verification of an implementation given in a resource-constraint-oriented specification against an architecture given in a constraint-oriented specification.

   process *Q_entity* [*Q,CLQ*]: **noexit** :=
   QA_local [*Q*] |[*Q*]| (*Q_send* [*Q,CLQ*]||| *Q_receive* [*Q,CLQ*])
   where
     process *QA_local* [*X*]: **noexit** :=
     X?*x:question*; *X*?*y:answer*; **stop**
     endproc
     process *Q_send* [*X, Y*]: **noexit** :=
     X?*x:question*; *Y*!*send*!*encode_q(x)*; **stop**
     endproc
     process *Q_receive* [*X, Y*]: **noexit** :=
     Y!*receive*?*d:data*; *X*!*decode_a(d)*; **stop**
     endproc
   endproc

Example 4.3 (*state-oriented specification of Q_entity*). We may derive a subset of the states of the protocol entity *Q_entity* by representing the global states of the *QA_service* as pairs of local states, as discussed in Example 3, and extract the local states that can be attributed to the protocol entity located at gate *Q*. The remaining

states of *Q_entity* are related to the interactions of *Q_entity* with the lower level service. Note that *Q_entity* should be instantiated in *QA_service* as *Q_entity* [*Q,CLQ*] (*await_q, empty, empty*).

The example given below corresponds to the traditional approach of describing protocol entities as state machines. The state table descriptions of OSI protocols are an example of this approach.

> **process** Q_entity [Q,CLQ] (s:local_state, b1:q_buffer, b2:a_buffer): **noexit** :=
>   [s = await_q] → Q?x:question;
>   Q_entity [Q,CLQ] (pending_q, push_q(x, b1), b2)
> ⫴ [s = pending_q] → CLQ!send!encode_q(top_q(b1));
>   Q_entity [Q,CLQ] (await_a, b1, b2)
> ⫴ [s = await_a] → CLQ!receive?d:data;
>   Q_entity [Q,CLQ] (pending_a, b1, push_a(decode_a(d), b2)
> ⫴ [s = pending_a] → Q!top_a(b2);
>   Q_entity [Q,CLQ] (done_q, b1, b2)
> ⫴ [s = done_q] → **stop**
> **endproc**

Relation to language constructs

As with the constraint-oriented style, the resource-oriented style induces extensive usage of parallel composition operators. Unlike the constraint-oriented style, hiding is used for internal interactions to make them invisible in the external behaviour.

4. Verification of examples

The previous examples demonstrate how several styles may be used along the design trajectory of a distributed system in order to support the design principles and objectives. At each stage the specification builds on the previously developed structure, thus permitting a step-by-step development of a final specification that can be used as a reference for product implementation. Correctness of the implementation requires that the desired external behaviour as defined by the initial specification is preserved in some appropriate sense by each design step.

In this section we discuss the verification of two steps in the design of the question/answer service and show the role of the specification styles. In particular we show that

(1) the constraint-oriented specification of the question/answer service (Example 2) is equivalent to the monolithic specification (Example 1) and

(2) the resource-oriented specification of the question/answer service (Example 4) is equivalent to (i.e. the protocol is correct with respect to) the constraint-oriented specification.

We will only consider the verification of dynamic behaviour, whereas a complete treatment would require additional reasoning concerning the role of the data types that are used. Standard theory on the verification of abstract data types can be found in, for example, [10, 11]. This limitation means that the specifications presented in Section 3 can be simplified by replacing the events that model the communication of data with events that model merely synchronization (each event consists of only a gate name). Alternatively, one can view the simplification as reducing the number of values per data sort to one.

With this simplification, and the introduction of new, short names for processes and gates, the question/answer service can be represented in the following ways:

● monolithic specification:

process *Sm[Qq, Aq, Aa, Qa]: noexit := Qq; Aq; Aa; Qa;* **stop endproc**

● constraint-oriented specification:

process *Sc[Qq, Aq, Aa, Qa]: noexit*
:= (L[Qq, Qa] ||| L[Aq, Aa]) || Rc[Qq, Aq, Qa, Aa]
where
process *L[Xq, Xa]: noexit := Xq; Xa;* **stop endproc**
process *Rc[Qq, Aq, Qa, Aa]: noexit := Qq; Aq;* **stop** *||| Aa; Qa;* **stop endproc**
endproc

● resource-oriented specification:

process *Sr[Qq, Aq, Aa, Qa]: noexit :=*
 hide UQs, UQr, UAs, UAr **in**
 ((QE[Qq, Qa, UQs, UQr] ||| AE[Aq, Aa, UAs, UAr]) |[UQs, UQr, UAs, UAr]|
 US[UQs, UQr, UAs, UAr])
where
process *QE[Qq, Qa, UQs, UQr]: noexit := L[Qq, Qa] |[Qq, Qa]|*
 (QEs[Qq, UQs] ||| QEr[Qa, UQr]) **endproc**
process *AE[Aq, Aa, UAs, UAr]: noexit := L[Aq, Aa] |[Aq, Aa]|*
 (AEs[Aa, UAs] ||| AEr[Aq, UAr]) **endproc**
process *US[UQs, UQr, UAs, UAr]: noexit*
 := UQs; UAr; **stop** *||| UAs; UQr;* **stop endproc**
process *QEs[Qq, UQs]: noexit := Qq; UQs;* **stop endproc**
process *QEr[Qa, UQr]: noexit := UQr; Qa;* **stop endproc**
process *AEs[Aa, UAs]: noexit := Aa; UAs;* **stop endproc**
process *AEr[Aq, UAr]: noexit := UAr; Aq;* **stop endproc**
endproc

4.1. Equivalence of behaviours: a few laws

In the literature there are many different notions of equivalence of observable behaviour. The LOTOS standard [17] contains definitions of and laws for two well-established notions of observational equivalence, viz. weak bisimulation equivalence (cf. [24, 22, 1]) and testing equivalence (cf. [8, 4, 9]). Of the two, weak bisimulation is the strongest, i.e. if two behaviours are weak bisimulation equivalent, they are also testing equivalent, but not necessarily vice versa.

We will show that the specifications given above are weak bisimulation equivalent, therefore also testing equivalent. To this end we make use of some of the laws for weak bisimulation congruence (i.e. equivalence that is preserved in every specification context, notation "=") that are stated in the LOTOS standard and of a few additional theorems which are proven below. For the proofs we apply to LOTOS the notion of (strong) bisimulation equivalence [24] (notation "~"), which is stronger than weak bisimulation congruence: $B1 \sim B2$ iff a relation R over behaviour expressions exists with $\langle B1, B2 \rangle \in R$, such that $\forall \langle A, B \rangle \in R$ and $\forall g \in G \cup \{i, \delta\}$, where G is the set of user-definable gates

(i) $A \xrightarrow{g} A' \Rightarrow \exists B'. B \xrightarrow{g} B'$ and $\langle A', B' \rangle \in R$,

(ii) $B \xrightarrow{g} B' \Rightarrow \exists A'. A \xrightarrow{g} A$ and $\langle A', B' \rangle \in R$.

Theorem. $stop |[S]| B \sim stop$ if $L(B) \subseteq S$ and B is stable (*i.e. cannot perform internal transitions*: $B \xrightarrow{g} B' \Rightarrow g \neq i$).

Proof. The proof follows immediately from the operational semantics of the LOTOS parallel operator, as defined by the inference rules that express the transition possibilities of a behaviour expression $B1 |[S]| B2$:

$$B1 \xrightarrow{g} B1' \text{ and } g \in (L(B1) - S) \cup \{i\} \Rightarrow B1 |[S]| B2 \xrightarrow{g} B1' |[S]| B2,$$

$$B2 \xrightarrow{g} B2' \text{ and } g \in (L(B2) - S) \cup \{i\} \Rightarrow B1 |[S]| B2 \xrightarrow{g} B1 |[S]| B2',$$

$$B1 \xrightarrow{g} B1' \text{ and } B2 \xrightarrow{g} B2' \text{ and } g \in (L(B1) \cap L(B2) \cap S) \cup \{\delta\}$$

$$\Rightarrow B1 |[S]| B2 \xrightarrow{g} B1' |[S]| B2'. \qquad \square$$

Theorem (associativity of parallel composition).

$$(A |[S1]| B) |[S2]| C \sim A |[S1]| (B |[S2]| C)$$

$$\textit{if } L(A) \cap S2 \subseteq L(A) \cap S1 \text{ and } L(C) \cap S1 \subseteq L(C) \cap S2.$$

Proof. We define the following abbreviations:

$$B1 \stackrel{\text{def}}{=} (A \,|[\,S1\,]|\, B) \,|[\,S2\,]|\, C, \qquad B2 \stackrel{\text{def}}{=} A \,|[\,S1\,]|\, (B \,|[\,S2\,]|\, C) \quad \text{and}$$

$$R \stackrel{\text{def}}{=} \{\langle (E \,|[\,S1\,]|\, F) \,|[\,S2\,]|\, G, E \,|[\,S1\,]|\, (F \,|[\,S2\,]|\, G) \rangle$$

$$|\, E, F, G \text{ behaviour expressions}\}.$$

From the definition of R it follows that $\langle B1, B2 \rangle \in R$. We enumerate the potential transition cases of $B1$ and $B2$ in terms of transitions of A, B and C, and state the conditions such that $B1 \stackrel{g}{\rightarrow} B1' \Rightarrow \exists B2'. B2 \stackrel{g}{\rightarrow} B2'$ and $B2 \stackrel{g}{\rightarrow} B2' \Rightarrow \exists B1'. B1 \stackrel{g}{\rightarrow} B1'$. The cases and conditions follow from the inference rules defining the operational semantics of the parallel operator:

(a) $A \stackrel{g}{\rightarrow} A'$: $L(A) - (S1 \cup S2) = L(A) - S1 \Leftrightarrow L(A) \cap S2 \subseteq L(A) \cap S1$;

(b) $B \stackrel{g}{\rightarrow} B'$: no conditions, i.e. transitions of $B1$ and $B2$ depend in the same way on $L(A)$, $L(B)$, $L(C)$, $S1$ and $S2$;

(c) $C \stackrel{g}{\rightarrow} C'$: $L(C) - S2 = L(C) - (S1 \cup S2) \Leftrightarrow L(C) \cap S1 \subseteq L(C) \cap S2$;

(d) $A \stackrel{g}{\rightarrow} A'$ and $B \stackrel{g}{\rightarrow} B'$: no conditions;

(e) $A \stackrel{g}{\rightarrow} A'$ and $C \stackrel{g}{\rightarrow} C'$: $(L(A) \cap L(C) \cap S2) - S1 = (L(A) \cap L(B) \cap S1) - S2$.

This condition is only true if the left- and righthand side are equal to $\emptyset$ (in which case this transition cannot occur), which is implied by $L(A) \cap S2 \subseteq L(A) \cap S1$ and $L(C) \cap S1 \subseteq L(C) \cap S2$;

(f) $B \stackrel{g}{\rightarrow} B'$ and $C \stackrel{g}{\rightarrow} C'$: no conditions;

(g) $A \stackrel{g}{\rightarrow} A'$ and $B \stackrel{g}{\rightarrow} B'$ and $C \stackrel{g}{\rightarrow} C'$: no conditions.

It follows directly from the definition of the parallel operator that also $\langle B1', B2' \rangle \in R$. $\square$

Theorem (parallel composition reshuffling).

$$(A \,|[\,S1\,]|\, B) \,|[\,I1 \cup I2\,]|\, (C \,|[\,S2\,]|\, D) \sim (A \,|[\,I1\,]|\, C) \,|[\,S1 \cup S2\,]|\, (B \,|[\,I2\,]|\, D)$$

if $L(A) \cap (S2 \cup I2) = \emptyset$, $L(B) \cap (S2 \cup I1) = \emptyset$, $L(C) \cap (S1 \cup I2) = \emptyset$ *and* $L(D) \cap (S1 \cup I1) = \emptyset$.

Proof. We define

$$B1 \stackrel{\text{def}}{=} (A \,|[\,S1\,]|\, B) \,|[\,I1 \cup I2\,]|\, (C \,|[\,S2\,]|\, D),$$

$$B2 \stackrel{\text{def}}{=} (A \,|[\,I1\,]|\, C) \,|[\,S1 \cup S2\,]|\, (B \,|[\,I2\,]|\, D) \quad \text{and}$$

$$R \stackrel{\text{def}}{=} \{\langle (E \,|[\,S1\,]|\, F) \,|[\,I1 \cup I2\,]|\, (G \,|[\,S2\,]|\, H),$$

$$(E \,|[\,I1\,]|\, F) \,|[\,S1 \cup S2\,]|\, G \,|[\,I2\,]|\, H) \rangle |\, E, F, G, H$$

behaviour expressions$\}$.

Clearly, $\langle B1, B2 \rangle \in R$. We check whether for all possible transition cases of $B1$ and $B2$ the conditions that must apply if $B1 \stackrel{g}{\rightarrow} B1' \Rightarrow \exists B2'. B2 \stackrel{g}{\rightarrow} B2'$ and $B2 \stackrel{g}{\rightarrow} B2' \Rightarrow \exists B1'. B1 \stackrel{g}{\rightarrow} B1'$ are implied by the conditions of the theorem:

(a) $A \stackrel{g}{\rightarrow} A'$: $L(A) - (S1 \cup I1 \cup I2) = L(A) - (S1 \cup S2 \cup I1)$. This is true since $L(A) \cap (S2 \cup I2) = \emptyset$;

(b) $B \stackrel{g}{\rightarrow} B'$: $L(B) - (S1 \cup I1 \cup I2) = L(B) - (S1 \cup S2 \cup I2)$. True, since $L(B) \cap (S2 \cup I1) = \emptyset$;

(c) $C \xrightarrow{g} C'$: $L(C) - (S2 \cup l1 \cup l2) = L(C) - (S1 \cup S2 \cup l1)$. True, since $L(C) \cap (S1 \cup l2) = \emptyset$;

(d) $D \xrightarrow{g} D'$: $L(D) - (S2 \cup l1 \cup l2) = L(D) - (S1 \cup S2 \cup l2)$. True, since $L(D) \cap (S1 \cup l1) = \emptyset$;

(e) $A \xrightarrow{g} A'$ and $B \xrightarrow{g} B'$: $(L(A) \cap L(B) \cap S1) - (l1 \cup l2) = (L(A) \cap L(B) \cap (S1 \cup S2)) - (l1 \cup l2)$. True, since $L(A) \cap S2 = \emptyset$;

(f) $A \xrightarrow{g} A'$ and $C \xrightarrow{g} C'$: $(L(A) \cap L(C) \cap (l1 \cap l2)) - (S1 \cup S2) = (L(A) \cap L(C) \cap l1) - (S1 \cup S2)$. True, since $L(A) \cap l2 = \emptyset$;

(g) $B \xrightarrow{g} B'$ and $D \xrightarrow{g} D'$: $(L(B) \cap L(D) \cap (l1 \cup l2)) - (S1 \cup S2) = (L(B) \cap L(D) \cap l2) - (S1 \cup S2)$. True, since $L(B(\cap l1 = \emptyset$;

(h) $C \xrightarrow{g} C'$ and $D \xrightarrow{g} D'$: $(L(C) \cap L(D) \cap S2) - (l1 \cup l2) = (L(C) \cap L(D) \cap (S1 \cup S2)) - (l1 \cup l2)$. True, since $L(C) \cap S1 = \emptyset$;

(i) $A \xrightarrow{g} A'$ and $B \xrightarrow{g} B'$ and $C \xrightarrow{g} C'$ and $D \xrightarrow{g} D'$:

$$(L(A) \cap L(B) \cap L(C) \cap L(D) \cap S1 \cap S2 \cap (l1 \cup l2)) \cup \{\delta\}$$

$$= (L(A) \cap L(B) \cap L(C) \cap L(D) \cap (S1 \cup S2) \cap (l1 \cap l2)) \cup \{\delta\}.$$

True, since $L(A) \cap (S2 \cup l2) = \emptyset$.

Other transition cases are not possible for *B1* and *B2* with the conditions of the theorem. Transitions of *B1* (*B2*, respectively) require a *g* action with $g \in l1 \cup l2$ ($g \in S1 \cup S2$), while the parts involved are such that some have no gates in common with *l1* (*S1*) and the others have no gates in common with *l2* (*S2*).

It follows directly from the definition of the parallel operator that also $\langle B1', B2' \rangle \in R$. □

Table 1 summarizes the weak bisimulation congruence laws which will be used for the verification exercise. Of the above theorems ((9), (10) and (11) in the table) also some interesting special cases are indicated ((10a), (10b) and (11a)).

4.2. Verification of the constraint-oriented specification

Verifying the correctness of the constraint-oriented specification of the question/answer service with respect to the monolithic specification is straightforward, since it requires application of the laws for instantiation and expansion only. In the following we show the transformation of *Sc* to *Sm*. We justify each transformation step with a reference to the law of Table 1 which is made use of.

$Sc[Qq, Aq, Aa, Qa]$
$= (Qq; Qa; \textbf{stop} \, ||| \, Aq; Aa; \textbf{stop}) \, || \, (Qq; Aq; \textbf{stop} \, ||| \, Aa; Qa; \textbf{stop})$ (5)
$= Qq; ((Qa; \textbf{stop} \, ||| \, Aq; Aa; \textbf{stop}) \, || \, (Aq; \textbf{stop} \, ||| \, Aa; Qa; \textbf{stop}))$ (7)
$= Qq; Aq; ((Qa; \textbf{stop} \, ||| \, Aa; \textbf{stop}) \, || \, (\textbf{stop} \, ||| \, Aa; Qa; \textbf{stop}))$ (7)
$= Qq; Aq; Aa; ((Qa; \textbf{stop} \, ||| \, \textbf{stop}) \, || \, (\textbf{stop} \, ||| \, Qa; \textbf{stop}))$ (7)
$= Qq; Aq; Aa; Qa; \textbf{stop}$ (7)
$= Sm[Qq, Aq, Aa, Qa]$ (5)

Table 1. Some useful laws of weak bisimulation congruence (without value expressions).

hiding

(1) **hide** S **in** $B = B$ if $L(B) \cap S = \emptyset$

(2) **hide** S **in** g; $B = i$; (**hide** S **in** B) if $g \in S$

(3) **hide** S **in** g; $B = g$; (**hide** S **in** B) if $g \notin S$

(4) **hide** S **in** $(B1 |[S']| B2) - ($**hide** S **in** $B1) |[S']| ($**hide** S **in** $B2)$ if $S \cap S' = \emptyset$

instantiation

(5) $b[a1, \ldots, an] = Bb[a1/g1, \ldots, an/gn]$ if **process** $b[g1, \ldots, gn]:f := Bb$ **endproc** is the format of the corresponding process abstraction for the process-identifier b

internal action

(6) a; i; $B = a$; B

expansion

Let $B1 \, [] \, B2 \, [] \cdots [] \, Bn$ be writtten as $[]\{B1, B2, \ldots, Bn\}$, with n finite and let $B = []\{bi; Bi \mid i \in I\}$ and $C = []\{cj; Cj \mid j \in J\}$, with I and J finite sets. Then

(7) $B \,|[S]| \, C = []\{bi; (Bi \,|[S]| \, C) \mid bi \notin S, i \in I\} \,[] \, []\{cj; (B \,|[S]| \, Cj \mid cj \notin S, j \in J\} \,[] \, []\{a; (Bi \,|[S]| \, Cj) \mid a = bi = cj, a \in S, i \in I, j \in J\}$

parallel composition

(8) $A \,|[S]| \, B = B \,|[S]| \, A$

(9) **stop** $|[S]| \, B =$ **stop** if $L(B) \subseteq S$ and B is stable

(10) $(A \,|[S1]| \, B) \,|[S2]| \, C = A \,|[S1]| \, (B \,|[S2]| \, C)$ if $L(A) \cap S2 \subseteq L(A) \cap S1$ and $L(C) \cap S1 \subseteq L(C) \cap S2$

(10a) $(A \,|[S1]| \, B) \,|[S2]| \, C = A \,|[S1]| \, (B \,|[S2]| \, C)$ if $L(A) \cap S2 = \emptyset$ and $L(C) \cap S1 = \emptyset$

(10b) $(A \,|[S]| \, B) \,|[S]| \, C = A \,|[S]| \, (B \,|[S]| \, C) = A \,|[S]| \, B \,|[S]| \, C$

(11) $(A \,|[S1]| \, B) \,|[I1 \cup I2]| \, (C \,|[S2]| \, D) \sim (A \,|[I1]| \, C) \,|[S1 \cup S2]| \, (B \,|[I2]| \, D)$ if $L(A) \cap (S2 \cup I2) = \emptyset$, $L(B) \cap (S2 \cup I1) = \emptyset$, $L(C) \cap (S1 \cup I2) = \emptyset$ and $L(D) \cap (S1 \cup I1) = \emptyset$

(11a) $(A \,|[S1]| \, B) \,|||\, (C \,|[S2]| \, D) = (A \,|||\, C) \,|[S1 \cup S2]| \, (B \,|||\, D)$ if $(L(A) \cup L(B)) \cap S2 = \emptyset$, $(L(C) \cup L(D)) \cap S1 = \emptyset$

4.3. Verification of the resource-oriented specification

In this case we exploit the structure of the given specifications in order to derive the equivalence of two specifications from the congruence of smaller (simpler) behaviours, thus reducing the verification effort. First, common parts in the specifications may be identified that may subsequently be eliminated if their composition with the remaining part is the same in both specifications. In other words, the equivalence of $C[B1]$ and $C[B2]$, where $C[\ ]$ is a context and $B1$ and $B2$ are behaviour expressions, is implied by the congruence of $B1$ and $B2$.

The above approach is a special case of that of identifying "corresponding" parts, i.e. behaviours that describe corresponding functionality and for that reason are expected to be equivalent. If one can identify such parts, it is natural to investigate the congruence of the pairs independently: the equivalence of $C[D1, \ldots, Dn]$ and $C[E1, \ldots En]$ is implied by the congruence of all the pairs $D1$ and $E1$, $D2$ and $E2$, etc.

The structures chosen for the specifications, and therefore also the specification styles adopted, determine how easy or difficult it is to find common/corresponding parts. As stated before, a constraint-oriented specification may provide hints for further (internal) structuring. Following these hints in the development of a resource-

oriented specification can be advantageous. That is illustrated in the following transformation of *Sr* to *Sc*.

Sr[Qq, Aq, Aa, Qa]
  = **hide** *UQs, UQr, UAs, UAr* **in**   (5)
    (((L[Qq, Qa] |[Qq, Qa]| (QEs[Qq, UQs]||| QEr[Qa, UQr]))
       ||| *(L[Aq, Aa] |[Aq, Aa]| (AEs[Aa, UAs]||| AEr[Aq, UAr]))*
)
    |[UQs, UQr, UAs, UAr]|
      US[UQs, UQr, UAs, UAr]
)
  = **hide** *UQs, UQr, UAs, UAs* **in**   (11a)
    (((L[Qq, Qa]||| L[Aq, Aa])
      |[Qq, Qa, Aq, Aa]|
        ((QEs[Qq, UQs]||| QEr[Qa, UQr])||| (AEs[Aa, UAs]||| AEr[Aq, UAr]))
)
    |[UQs, UQr, UAs, UAr]|
      US[UQs, UQr, UAs, UAr]
)
  = **hide** *UQs, UQr, UAs, UAr* **in**   (10a)
    ((L[Qq, Qa]||| L[Aq, Aa])
    |[Qq, Qa, Aq, Aa]|
      (((QEs[Qq, UQs]||| QEr[Qa, UQr])||| (AEs[Aa, UAs]||| AEr[Aq, UAr]))
      |[UQs, UQr, UAs, UAr]|
        US[UQs, UQr, UAs, UAr]
))
  = *(***hide** *UQs, UQr, UAs, UAr* **in** *(L[Qq, Qa]||| L[Aq, Aa]))*   (4)
    |[Qq, Qa, Aq, Aa]|
      *(***hide** *UQs, UQr, UAs, UAr* **in**
        (((QEs[Qq, UQs]||| QEr[Qa, UQr])||| (AEs[Aa, UAs]||| AEr[Aq, UAr]))
        |[UQs, UQr, UAs, UAr]|
          US[UQs, UQr, UAs, UAr]
))
  = *(L[Qq, Qa]||| L[Aq, Aa])*   (1)
    |[Qq, Qa, Aq, Aa]|
      *(***hide** *UQs, UQr, UAs, UAr* **in**
        (((QEs[Qq, UQs]||| QEr[Qa, UQr])
          ||| (AEs[Aa, UAs]||| AEr[Aq, UAr]))
        |[UQs, UQr, UAs, UAr]|
          US[UQs, UQr, UAs, UAr]
))

We observe that the latter behaviour expression has its first part in common with the expression that defines *Sc* (the part representing the local behaviour at the

interfaces). Therefore, if we can prove the congruence of the remaining part of both expressions (representing end-to-end behaviour) we are done. Let the protocol end-to-end behaviour, i.e. the last hide-expression above, be denoted by *Rr*. Since the service end-to-end behaviour is denoted by *Rc*, we have to show the transformation of *Rr* into *Rc*. To shorten the presentation of the proof, we allow the application of more than one law in a transformation step.

$Rr[Qq, Aq, Qa, Aa]$
 $=$ **hide** UQs, UQr, UAs, UAr **in**   (5, 10b, 8)
  $(\ ((QEs[Qq, UQs] ||| AEr[Aq, UAr]) ||| (QEr[Qa, UQr] ||| AEs[Aa, UAs]))$
  $|[UQs, UQr, UAs, UAr]|$
   $(UQs; UAr;$ **stop** $||| UAs; UQr;$ **stop**$)$
  $)$
 $=$ **hide** UQs, UQr, UAs, UAr **in**   (11a)
  $(\ ((QEs[Qq, UQs] ||| AEr[Aq, UAr] |[UQs, UAr]| UQs; UAr;$ **stop**$)$
  $|||$
  $((QEr[Qa, UQr] ||| AEs[Aa, UAs]) |[UQr, UAs]| UAs; UQr;$ **stop**$)$
  $)$
 $=$ **(hide** UQs, UQr, UAs, UAr **in**   (4, 5)
   $((Qq; UQs;$ **stop** $||| UAr; Aq;$ **stop**$) |[UQs, UAr]| UQs; UAr;$ **stop**$))$
  $|||$ **(hide** UQs, UQr, UAs, UAr **in**
   $((UQr; Qa;$ **stop** $||| Aa; UAs;$ **stop**$) |[UQr, UAs]| UAs; UQr;$ **stop**$))$
 $= Qq;$ **(hide** UQs, UQr, UAs, UAr **in**   (7, 3)
   $((UQs;$ **stop** $||| UAr; Aq;$ **stop**$) |[UQs, UAr]| UQs; UAr;$ **stop**$))$
  $|||Aa;$ **(hide** UQs, UQr, UAs, UAr **in**
   $((UQr; Qa;$ **stop** $||| UAs;$ **stop**$) |[UQr, UAs]| UAs; UQr;$ **stop**$))$
 $= Qq; i;$ **(hide** UQs, UQr, UAs, UAr **in**   (7, 2)
   $(($ **stop** $||| UAr; Aq;$ **stop**$) |[UQs, UAr]| UAr;$ **stop**$))$
  $||| Aa; i;$ **(hide** UQs, UQr, UAs, UAr **in**
   $((UQr; Qa;$ **stop** $|||$ **stop**$) |[UQr, UAs]| UQr;$ **stop**$))$
 $= Qq; i; i;$ **(hide** UQs, UQr, UAs, UAr **in**   (7, 2)
   $(($ **stop** $||| Aq;$ **stop**$) |[UQs, UAr]|$ **stop**$))$
  $||| Aa; i; i;$ **(hide** UQs, UQr, UAs, UAr **in**
   $((Qa;$ **stop** $|||$ **stop**$) |[UQr, UAs]|$ **stop**$))$
 $= Qq; i; i; Aq;$ **stop** $||| Aa; i; i; Qa;$ **stop**   (7, 3, 9)
 $= Rc[Qq, Aq, Qa, Aa]$ (6, 5)

The sequence of the transformation steps above is suggested by the fact that it may be possible to find expressions corresponding to the independent behaviours (viz. those describing different directions of transfer) that form the end-to-end constraints of the constraint-oriented specification. This is indeed possible, and the transformation to *Rc* then is simple.

It appears that law (11) plays a crucial role in the verification. First it is used to separate the local constraints from the end-to-end constraints in the composite behaviour of the two protocol entities and the underlying service and subsequently it allows to separate the constraints associated with the two directions of transfer in the protocol end-to-end behaviour. We may wonder whether this approach is still viable if the specifications are not as simple as in the present case. For instance, in the constraint-oriented specification the end-to-end constraints for the two directions of transfer may be interrelated (this is the case, for example, in the OSI session service specification [32]) and similarly with the send and receive actions of a protocol entity in the resource-oriented specification (this is normally the case). It appears that the above mentioned approach indeed applies to more complex cases. The interested reader is referred to [33] where more general constraint- and resource-oriented specification structures are assessed in terms of their suitability for verification.

A final remark should be made about the internal structuring of a system (e.g., the question/answer service). In Section 3 it was mentioned that, in principle, this can be done in an infinite number of ways. Laws (2) and (6), in particular, illustrate how different compositions of a system may provide equivalent behaviour. Law (2) states that actions at internal gates, can be replaced by internal events in the behaviours of the compositions; law (6) states that processes with certain internal events cannot be observably distinguished from the same processes without those internal events (there are also laws that relate to internal actions in other contexts). Hence, application of these laws makes the particular internal structures expressed by different (e.g. resource-oriented) specifications disappear and allows them to be transformed, with the help of the other laws, into the same (e.g. constraint-oriented) specification.

5. Evaluation

In introducing the role of architecture we stated that abstractness is the most relevant criterion to evaluate its adequacy. The relevance of structuring abstract specifications was argued both from the standpoint of the system user to provide a comprehensible specification and from the standpoint of the system implementor to produce a reliable and effective implementation. A need for specification and implementation *methods* was thus recognized, leading to the identification of four distinct specification styles, whose relation to general design principles such as orthogonality, generality and open-endedness was addressed. These styles emerge from our own specification experience as practical approaches which can be used to serve different design objectives. It should be noted that these styles are actually extremes in the sense that specifications of "real" systems will normally be based on a mixture of these styles. Section 3.4 already exemplified such a mixture. Also other styles may prove worthwhile. Further research into the identification and characterization of specification styles is therefore encouraged.

The use of well-suited specification styles will be most effective if they are applied throughout the design trajectory, from the early conceptual phases of architectural design to the final phases of implementation development. If the formal description of an object can only start after it has been completely informally defined, see e.g. the current situation of OSI standards, the chance to produce high-quality architectures and implementations is much reduced. On the other hand, one cannot expect a team of industrial designers to become familiar with a specification discipline overnight. One should rather anticipate a necessary habituation period, after which specification styles can be applied routinely. The tuned interworking of styles in the system design trajectory will only prove effective if it facilitates the design process and improves the quality of its result. We evaluate now to which extent the four styles presented in Section 3 may serve these purposes.

5.1. Requirements capturing: the constraint-oriented style

During the first phase of design, the system achitecture is best defined using an extensional description. If the architecture is extremely simple, the monolithic style is best suited for its specification. Such a specification is then the most compact one and poses no problems with respect to comprehensibility. Fairly complex architectures, however, require a structured approach in order to find comprehensible presentations. The constraint-oriented style is best suited for this purpose since it enables the human understanding of the architecture and reasoning about it, in terms of the user requirements on the system's external behaviour and the relations between these requirements.

It is also worth mentioning that, especially in the early phases of formation of the user requirements, inconsistencies, errors, or unfeasible prescriptions are likely to occur. Forcing a formal specification of user requirements then proves extremely beneficial to their—possibly machine-aided—analysis, and helps to prevent, or detect deficiencies at an early stage. Undetected deficiencies propagate throughout the subsequent stages of the design trajectory at unpredictable costs.

Requirements structuring is more art than science. Yet, not only the economical and strategic relevance of this problem is generally recognized (see e.g. the recommendations in [31], striving for adoption of FDTs at the early phases of standardization projects), but also the awareness of its scientific relevance is penetrating the "ivory tower" of theoretical circles. For instance, in [26] the problem of structuring specifications is considered to be most important and considerable attention is paid to specification styles in algebraic specification.

It is reasonable to evaluate the implementation-oriented styles on the basis of their adequacy for developing and validating implementations. This evaluation involves technical criteria, such as the ease with which subprocesses can be mapped on implementation building-bricks. The various state-machine models that underly the state-oriented approacch are well-researched, allowing for quite direct implementation strategies, as well as different validation methods. The resource-oriented style has a less developed theory because the concept of state is easier to

formalize than that of a resource. Consequently, the success of the application of this style will be judged mainly on the basis of experience and pragmatic arguments.

The constraint-oriented style, on the other hand, should be evaluated on the basis of its adequacy to capture specification requirements. In the case of LOTOS and other process-algebraic formalisms such as TCSP [7] or CIRCAL [23], the mechanism that is used to achieve this can be explained in a formal setting. As this mechanism is less known than it should be, we digress shortly on this topic.

Parallel composition as logical conjunction

The usefulness of the constraint-oriented style should be appreciated in the light of the constructive nature of LOTOS and related process-algebraic formalisms. In logical languages, many of which are nonconstructive par excellence, a constraint-oriented style is dictated by the formalism, but generally at the cost of a greater gap between specification and implementation. In [13] it is explained how some of the constraint-oriented power of logical languages may be salvaged for constructive techniques. The main point is that parallel composition may be used to implement a logical conjunction. Although this concerns only one logical operator it is most important: many systems are conceived of as models of the conjunction of a large set of requirements. This implies an important structuring principle: each requirement is expressed by a separate process and these processes are composed in parallel. It is clear that this technique can be applied iteratively, also decomposing the processes that correspond to requirements that can be factorized as a conjunction of simpler requirements, etc. Open-endedness is supported in a straightforward manner: adding a requirement corresponds to the addition of (a) process(es).

To express how the representation of conjunction works we use the assertion $B \, \text{sat}_A \, P$, where B is a behaviour expression, A a set of gates and P a predicate. $B \, \text{sat}_A \, P$ is true if and only if the restrictions to A of all elements of the (prefix-closed) set of observable traces of B satisfy property P. The precise way in which parallel composition can support the expression of conjunction is indicated by the following lemma.

Lemma. *If $B_1 \, \text{sat}_A \, P$ and $B_2 \, \text{sat}_A \, Q$ then $B_1 \, |[A]| \, B_2 \, \text{sat}_A \, P \wedge Q$.*

The proof is an adaptation of the work in [13] and can be found in [5, 6]. It can be modified in various ways, e.g. when the gates that are common to B_1 and B_2 are all in A. Also, in the above only properties of trace sets are dealt with, which suffices for dealing with *safety* properties, but does not guarantee the progress of behaviour, or *liveness*. In part this can be remedied by use of *failures*, i.e. traces + refusal sets, and using a more complicated satisfaction relation [13].

It is also possible to model other logical operators in constructive techniques, most notably disjunction (e.g. by a choice construct), but this seems to be of less decisive importance for the success of constraint-oriented specification.

5.2. System implementation: from styles to methodology

Intensional descriptions support the implementation of the system achitecture. Even if the extensional description was a simple one, say in the monolithic style, this does not mean that the system is easy to implement. The distribution of the resources available for the implementation may present technical difficulties that can only be overcome by introducing internal functions which were not visible in the system architecture. A resource-oriented style is best suited for this purpose since it enables the description of internal structures in a straightforward way. Each of the devised resources may again be described in any of the four styles. This iterative integration of different styles in the system design trajectory, see Fig. 3, forms the basis of a design method.
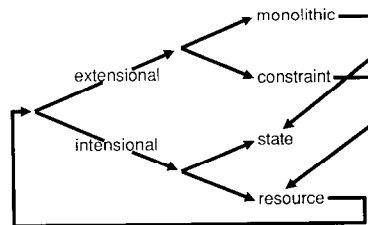

Fig. 3. Related specification styles.

Specifications in the resource-constraint-oriented style may benefit from a constraint-oriented specification by reusing part of the latter. A state-oriented specification style should be viewed as the last step in the formal design trajectory and should only be undertaken when no further substructuring of the system or resource is required. Such specifications are best derived from monolithic ones. This style is most restrictive: implementors can only directly copy the states and state transitions in the implementation, e.g. by using table look-up techniques, unless they are willing to redesign the system.

The constraint-oriented and resource-oriented styles can be used iteratively until the abstract resources are simple enough to allow some easy mapping of their descriptions and arrangements onto implementation constructs. These styles are to be preferred for intermediate specifications along the design trajectory, since they best match advanced implementation techniques which aim at (partially) automated implementation. In the case of complex systems, these styles support general design principles which should be adhered to for producing high quality architectures and implementations.

5.3. Specification styles and specification languages

Our final point of evaluation concerns the extent to which different styles are supported by given specification formalisms. In Section 3 the four styles have been

characterized in general terms and exemplified with specifications in LOTOS. According to [27], specification languages can be characterized by how extensive the state component of the underlying model is versus their ability to define allowable state transitions, or interaction sequences. Some languages may therefore be more suitable to express state-oriented specifications or monolithic specifications and less suitable to support mixtures of styles, including styles that favour structuring. Consequently, transformations between specifications in different styles may prove not possible in a given language.

LOTOS is considered a broad-spectrum specification language which supports the four specification styles mentioned in this paper. It is interesting to note that the importance of constraint-oriented specification for the implementation-independent description of open systems has caused significant feedback on the design of LOTOS. During the development of the LOTOS standard it has led to a change in the definition of parallel composition and more recently it has led to the development of Extended LOTOS [5], whose syntax and semantics have been optimized with respect to the application of this style.

6. Conclusions

In this paper we have argued that in practice an architecture needs to be structured, thus possibly influencing implementation choices, despite the fact that in principle an architecture is an implementation-independent definition of externally observable behaviour. This implies a responsibility of the architect for the quality of the implementations. To exploit this responsibility the architect should obey qualitative design principles like orthogonality, generality, and open-endedness. One way to obey such principles is by applying specification styles. We have introduced a set of specification styles. We have shown that such styles can be used in relationship to each other, as elements of a method that supports the complete design trajectory from requirement capturing to implementation, where at each level in this trajectory different design objectives are supported. The suitability of these styles to various aspects of the verification problem has been exemplified. The specification language LOTOS has been adopted as a vehicle to convey ideas and to present examples. This choice should not be viewed as a limitation of the applicability of specification styles. In [30], for example, it is shown that some of our styles can easily be determined in the very general framework of equational type logic [21].

ST 2304 (Lotosphere). The authors like to thank Luigi Logrippo for his useful comments on a previous version of the examples presented in this paper.

References

[1] S. Abramsky, Observation equivalence as a testing equivalence, *Theoret. Comput. Sci.* **53** (1987) 225–241.

[2] G.A. Blaauw and F.P. Brooks, Computer architecture, Vol. 1: Design decisions, Univ. of North-Carolina, 1989.

[3] T. Bolognesi and E. Brinksma, Introduction to the ISO specification language LOTOS, *Comput. Networks ISDN Systems* **14** (1) (1987) 25–59.

[4] E. Brinksma, G. Scollo and C. Steenbergen, LOTOS specifications, their implementations and their tests, in: *Proc. IFIP WG6.1, Protocol Specification, Testing, and Verification VI*, Montreal, Canada, (1987) 349–360.

[5] E. Brinksma, On the design of extended LOTOS, Doctoral Dissertation, Univ. of Twente, 1988.

[6] E. Brinksma, Constraint-oriented specification in a constructive specification technique, in: *Proc. Rex Workshop on Stepwise Refinement of Distributed Systems*, Mook, The Netherlands, 1989, Lecture Notes in Computer Science, Vol. 430 (Springer, Berlin, 1990) 130–152.

[7] S.D. Brooks, C.A.R. Hoare and A.W. Roscoe, A theory of communicating sequential processes, *J. Assoc. Comput. Mach.* **31** (1984) 560–599.

[8] R. De Nicola and M.C.B. Hennessy, Testing equivalences for processes, *Theoret. Comput. Sci.* **34** (1984) 83–133.

[9] R. De Nicola, Extensional equivalences for transition systems, *Acta Inform.* **24** (1987) 211–237.

[10] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1* (Springer, Berlin, 1985).

[11] J.A. Goguen, J.W. Thatcher and E.G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in: R. Yeh ed., *Current Trends in Programming Methodology IV* (Prentice-Hall, Englewood Cliffs, 1978) 80–149.

[12] R. Gotzhein, Specifying abstract types with LOTOS, in: *Proc. IFIP WC6.1, Protocol Specification, Testing, and Verification VI*, Montreal, Canada (1987) 13–22.

[13] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, Englewood Cliffs, 1985).

[14] ISO, *Basic Reference Model for Open Systems Interconnection*, Int. Standard ISO 7498 (ISO, 1984).

[15] ISO, *Transport Service Definition*, Int. Standard ISO 8072 (ISO, 1985).

[16] ISO, *Basic Connection Oriented Session Service Definition*, Int. Standard ISO 8326 (ISO, 1987).

[17] ISO, *LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, Int. Standard ISO 8807 (ISO, 1989).

[18] ISO, *OSI Service Conventions*, Techn. Report ISO/TR 8509 (ISO, 1987).

[19] ISO, Proposal for a New Work Item: Architectural Semantics for Formal Description Techniques, ISO/IEC JTC1/N58, Oct. 1987.

[20] J. v.d. Lagemaat and G. Scollo, On the use of LOTOS for the formal description of a transport protocol, in: *Proc. FORTE 88, Formal Description Techniques I*, Stirling, Scotland, (1989) 247–261.

[21] V. Manca, A. Salibra and G. Scollo, Equational type logic, *Theoret. Comput. Sci.* **77** (1990) 131–159.

[22] R. Milner, Calculi for synchrony and asynchrony, *Theoret. Comput. Sci.* **25** (1983) 267–210.

[23] G.J. Milne, CIRCAL and the representation of communication, concurrency, and time, *ACM TOPLAS* **7** (2) (1985) 270–298.

[24] D. Park, Concurrency and automata on infinite sequences, in: *Proc. 5th GI Conference*, Lecture Notes in Computer Science, Vol. 104 (Springer, Berlin, 1981).

[25] D.L. Parnas, On the criteria to be used in decomposing systems into modules, *Comm. ACM* **15** (12) (1972) 1053–1058.

[26] H. Reichel, *Initial Computability, Algebraic Specification, and Partial Algebras* (Clarendon Press, Oxford, 1987).

[27] R.L. Schwartz and P.M. Melliar-Smith, From state machines to temporal logic: specification methods for protocol standards, *IEEE Trans. Comm.* **30** (12) (1982) 2486–2496.

[28] G. Scollo, C.A. Visssers and A. Di Stefano, LOTOS in Practice, in: *Proc. IFIP 86, 10th World Congress,* Dublin, Ireland (1986) 869–875.

[29] G. Scollo and M. v. Sinderen, On the architectural design of the formal specification of the session standards in LOTOS, in: *Proc. IFIP WG6.1, Protocol Specification, Testing, and Verification VI,* Montreal, Canada (1987) 3–14.

[30] G. Scollo, On the use of equational type logic for software engineering and protocol design, in: *Proc. 1st Maghrebin Conf. on Artificial Intelligence and Software Engineering,* Constantine Algeria, (1989) 24–27.

[31] ISO/IEC-JTC1/SC21 Rapporteur for the special working group on strategic planning, call for contributions to a meeting during the SC 21 meetings in Florence on improving the quality of standards, ISO/IEC-JTC1/SC21/N3424, Jan. 1989.

[32] M. v. Sinderen, I. Ajubi and F. Caneschi, The application of LOTOS for the formal description of the ISO session layer, in: *Proc. FORTE 88, Formal Description Techniques I,* Stirling, Scotland (1989) 263–277.

[33] M. v. Sinderen, A verification exercise relating to specification styles in LOTOS, Univ. of Twente, Memorandum INF-89-18, March 1989.

[34] C.A. Vissers, Architectural requirements for the temporal ordering specification of distributed systems, in: *Proc. of EUTECO—European Teleinformatics Conference* (1983) 79–95.

[35] C.A. Vissers and G. Scollo, Formal specification of OSI, in: G. Muller, R. Blanc eds., *Networking in Open Systems, Proc. Int. Seminar,* Oberlech, Austria, Aug. 1986, Lecture Notes in Computer Science Vol. 248 (Springer, Berlin, 1987) 338–359.

[36] C.A. Vissers, G. Scollo and M. v. Sinderen, Architecture and specification style in formal descriptions of distributed systems, in: *Proc. IFIP WG6.1, Protocol Specification, Testing, and Verification VIII,* Atlantic City (1989) 189–204.

[37] C.A. Vissers, G. Scollo, R. Alderden, J. Schot and L. Ferreira Pires, The architecture of interaction systems, course notes Twente Univ., Enschede, Netherlands, March 1989.