*Kenneth J. Turner. Specification architecture illustrated in a communications context (pre-publication version). Computer Networks and ISDN Systems, 29(4):397-411, March 1997.*

# Specification Architecture Illustrated in a Communications Context

**Kenneth J. Turner**

Department of Computing Science and Mathematics

University of Stirling

Stirling FK9 4LA, Scotland

*Telephone +44-1786-467-420 Facsimile +44-1786-464-551 Email kjt@cs.stir.ac.uk*

**19 April 1996**

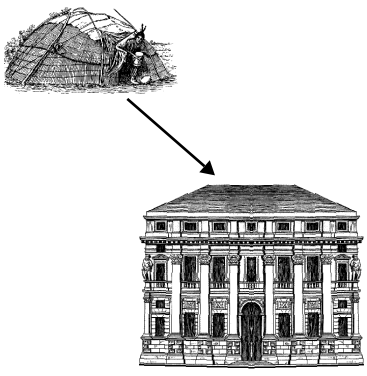
**Abstract**

The concept of specification architecture is introduced, and its importance is emphasised. Three important architectural principles are offered as a way of achieving a satisfactory specification architecture: modularity, generality and simplicity. These are supported by subsidiary techniques that realise them. Criteria are identified to evaluate successful application of the techniques and conformance to the principles. The approach is illustrated throughout with the example of a message switch, culminating in a larger case study based on the work of the PANGLOSS project to design a high-speed message switch.

**Keywords**: architecture, architectural principle, communications architecture, message switch, specification, specification architecture

# 1 Introduction

## 1.1 Specification Architecture



Except for completely trivial specifications, a specification necessarily has a significant structure – its **specification architecture**. Even non-constructive specifications have an architecture. The thesis of this paper is that an effective specification architecture is important and hard to achieve, but that there are general principles that can guide the specifier in choosing an appropriate architecture. Neglecting the architecture of a specification leads to the familiar woes of the software world: difficulty in understanding the specification, expense in modifying and extending it, problems in analysing it, etc. When a formal, mathematical specification is required, a good architecture is all but essential.

Specification architecture becomes particularly significant when the problem under study is ill-defined; in fact, finding a good architecture without actually specifying the problem is a major step

forwards. Experience shows that the major difficulty lies in *architecting* a specification, not *writing* it. By analogy, it is harder to *design* a program than to *code* it. Experience also shows that it may take several attempts to find a suitable architecture. It is tempting to tinker with the first architecture selected in the hope of making it match the problem better. However, it is usually better to throw away early attempts and to start again with a clearer understanding.

The quality of a specification architecture needs to be evaluated in the context of its use. For example, a specification may be organised differently depending on whether its purpose is to convey requirements clearly, to allow easy transition to an implementation, or to guide derivation of test cases. The level of technical sophistication expected of the reader might influence the style. The applications and even the development tools might affect the choice of architecture. Nonetheless, there are generally desirable features of specification architecture that apply in most cases.

Definitions of architecture in a standard dictionary emphasise style and structure: 'the art or science of building', 'a unifying or coherent form or structure', 'a method or style of building' [31]. Interestingly, definitions of 'specification' and 'formality' also emphasise form and structure, so there seems to be a link among all three words. The writings of famous historical architects (in the building sense) clearly indicate that architecture is a subtle combination of art and science. Certainly there is nothing mechanical and rule-based about choosing a specification architecture. Fortunately, there are sound principles that can be followed in the process of specification. The purpose of this paper is to elucidate some of these principles.

## 1.2 An Architectural Approach

The term 'architecture' is widely used (and perhaps abused) in computer science. A software architecture or a hardware architecture generally refers to a logical or physical design. Some authors make a sharp distinction between requirements and architecture, the latter being a complete design [29]. In the context of this paper an architecture is a structure, whether for specification or for design. Even more particularly, a specification architecture refers to the structure of a high-level requirements specification.

Many different architectures may be defined for the same system. For example the architectures may be defined at different stages during development and may thus have differing levels of abstraction (e.g. for requirements or implementation). The relationship between such architectures is one of refinement and is discussed in section Section 2.1. Another possibility is that the architectures reflect different viewpoints and so may not be strictly comparable. This occurs in ODP (Open Distributed Processing [14]), for example, where various viewpoints may be specified. Consistency among such architectures is still an open area of research [3].

An architecture may be purely syntactic and reflect the layout of the specification – textual or graphical. More significantly the structure may be semantic and be reflected in the behavioural organisation of the specification – interactions among elements,[1] hierarchical dependencies, or whatever. In general, the architecture of a specification is concerned with its elements and the way they are combined. This raises the question of whether a specification architecture is a design. The answer depends on what the structure represents. The conceptual architecture of a system, for example, should reflect only the structure inherent in the problem and should avoid solution bias. The implementation architecture, however, would be entirely focused on design aspects. It is thus necessary to be clear about the purpose of structure in an architecture.

Specification architecture may reflect the use of the specification language – its **style** [30]. If the specification language is sufficiently expressive it will be possible to adopt the conceptual architecture directly. However, a less suitable specification language or specification style might force a particular structure that is not intrinsic to the problem. Alternative ways of organising the specification may yield different benefits such as clarity, conciseness, comprehensibility, extensibility and analysability. Such specifications may differ only in essentially syntactic ways, produced by algebraic reformula-

[1]The term 'element' is preferred to 'component', since the latter suggests a concrete building block.

tion. However, much deeper variations may arise from substantial differences in the language constructs used, even if the resulting specifications are formally equivalent. Differences among such specification architectures might be considered 'horizontal' in that they do not concern changes in level of abstraction or non-determinism. It would not be usual to describe such structure as design.

Specification architecture may reflect some inherent or intended structure in the system being considered. The structure might be in the formulation of requirements, defining a framework for thinking about the system. The structure might be in the formulation of how the system should be built, defining a framework for constructing it. Specifications of the same thing may be written at many levels of abstraction, from fully abstract to fully concrete.[2] The more abstract (non-constructive) levels of specification may be considered as requirements, the more concrete (constructive) levels as implementations. The specification levels in between would be regarded as designs in the normal usage of the word. Differences among such specification architectures might be considered 'vertical' due to differing levels of abstraction or non-determinism.

Many aspects of architecture are specific to the problem domain in question. Since the combination of elements is a key feature of architecture, the combinators available in the specification language also have a major effect. For example, the combinators might be:

- sequence, iteration and selection in a program specification language
- conjunction, disjunction and negation in a logic specification language
- process creation, interleaving and synchronisation in a concurrent specification language.

Despite the dependency of specification architecture on problem domain and specification language, there is general architectural guidance that is relevant in most contexts, and this is the focus of the paper.

An architectural approach is needed to answer questions like:

- How can a good architecture be produced?
- Is *this* a good architecture?
- Is *this* a better architecture than *that*?

Behind such an approach are **architectural principles** that provide an overall framework. Such principles are put to work by using **architectural techniques** that realise them.

There is only a matter of degree between principles and techniques; it would be possible to regard techniques as subsidiary principles. At best, principles can only guide the architect. However, they can serve as a useful check-list of ideas to try. Architectural principles can be drawn from many disciplines; in software engineering these include systems theory, software design and formal methods. Communications system engineering builds on software engineering principles, and also on information theory, distributed system design and hardware engineering.

Enunciating principles will not guarantee their effective use. How can it be recognised that some principle has been properly applied? A complementary approach is therefore the use of **architectural criteria** to assess whether principles have been adhered to**.** Together, the principles (prescriptive) and the criteria (evaluative) provide a framework for the architect. There are many principles that could be adopted, so this paper concentrates on only a few important ones: modularity, generality and simplicity.

## 1.3 Related Work

Perhaps surprisingly, specification architecture appears to have attracted only limited attention in the literature. [16] is a classical discussion of important related issues. Specification architecture seems to have manifested itself mainly in the discussion of specification style. The closest work to that of this

[2]Making a specification more concrete involves adding design detail and removing non-determinism. Purists would regard only the most abstract (non-constructive) specification as a specification proper, and only the most concrete (constructive) specification as an implementation proper.

paper is on principles for design quality [19]. Although this work was rather different in context and was undertaken independently, the quality principles that are enunciated bear an uncanny resemblance to the principles and criteria discussed here. A good discussion is also presented in [19] of how quality principles can interact and even conflict. Intriguingly, there are parallels between specification structure and the structure and presentation of mathematical proofs [28].

In standards circles, work on an architectural semantics for OSI (Open Systems Interconnection, [9]) was undertaken in ISO/IEC project 97.43.1 [12]. Similar work is also being undertaken for ODP (Open Distributed Processing [14]) in Part 4 (Architectural Semantics [13]) of this work. [8] was motivated by distributed systems architecture, building on earlier specifications of architectural concepts using temporal logic [7].

Interestingly, much of the work on specification architecture has been stimulated by the use of just one formal approach: LOTOS (Language Of Temporal Ordering Specification [11]). The SEDOS project (Software Environment for the Design of Open distributed Systems, ESPRIT 410 [27]) included work on the architecture of LOTOS specifications. [23] built on this to define an architectural semantics for OSI using LOTOS. An investigation of architecture and specification style was also stimulated [30]. The SEDOS results were further developed by the PANGLOSS project discussed in Section 5; [2] and [18] record some of the results on specification architecture. The LOTOSPHERE project (ESPRIT 2304 [26]) extended treatment of architecture to the software life-cycle.

In the object-oriented world, architectural issues have been considered in the context of 'patterns' and 'frameworks' for design (e.g. [4, 5, 6]). It could be profitable to see what lessons can be learned from this for specification architecture in general.

## 1.4 Paper Structure

It might be difficult for the reader to relate to a purely theoretical exposition of architecting specifications. A simple but definite example has therefore been used throughout the paper – a message switch. The switch merely reads messages on one incoming line and routes them individually to one of a number of outgoing lines, as illustrated in Figure 1.

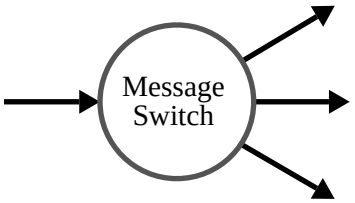



**Figure 1 Message Switch**

As each architectural principle and its criteria are introduced, examples of their use will be related to the message switch. Architectural criteria for the principle will then follow, together with further examples for the message switch. Sections 2, 3 and 4 of the paper deal respectively with the principles and criteria for modularity, generality and simplicity. Section 5 brings these to bear on a larger case study: a high-speed message switch. The principles, techniques and criteria to be studied are summarised in Table 1. Architecture principles, techniques and criteria are not absolute in any sense. The ones considered here might be regarded as less important or subsidiary by another architect (e.g. see the principles in [19]). The key thing is to employ *some* systematic architectural

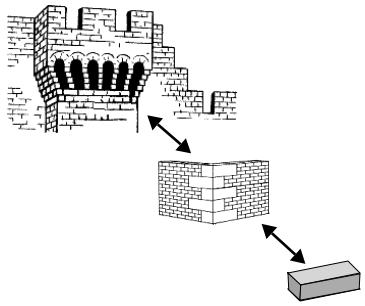
approach to specification.

| Principles | Techniques | Criteria |
|---|---|---|
| Modularity | Functional Decomposition<br>Constraint Decomposition<br>Temporal Decomposition<br>Spatial Decomposition | Coherence<br>Decoupling<br>Proportion |
| Generality | Parameterisation<br>Generalisation<br>Unification | Abstractness<br>Commonality<br>Adaptability |
| Simplicity | Idealisation<br>Deferment<br>Minimisation | Uniformity<br>Elegance<br>Economy |

**Table 1 Summary of Architectural Principles and Criteria**

# 2 Modularity

## 2.1 The Principle of Modularity



The principle of modularity is a familiar one; 'top-down specification', 'structured analysis' and 'divide and conquer' all apply the same approach. What may be less obvious is that there are many kinds of modularity apart from the obvious one of functional modularity, for example modularisation in terms of constraints, time or space. The objective is to have a tree (or possibly directed graph) of elements at progressive levels of detail. Such a tree has breadth (horizontal partitioning at the same level of abstraction) and depth (vertical partitioning at different levels of abstraction).

Modularity reflects a subdivision of the problem. The means of achieving this may be decomposition (top-down), composition (bottom-up), or even both. The end result is a structure of multiple hierarchical levels. Since the structure is more important here than the means of deriving it, decomposition is arbitrarily referred to in the following. However, similar observations can be made about composition.

**Functional decomposition** is commonly used on a complex system with rich functionality. To break this into manageable pieces it is normal to decompose the overall functionality into a hierarchy of functions. A wide range of so-called 'structured' methods adopt this approach. However, functions give but one view of a system.

**Constraint decomposition** is an alternative approach. Many systems are conveniently specified by separation of concerns. These concerns may be global and not related to just one function. In effect, the system may be projected onto a number of different 'planes', each giving a different but complementary view of the system. The overall specification is then the conjunction of these views. Some views may be distinct, others may overlap. Each view gives rules or constraints for some particular aspect of the system. This approach is assertional in nature, and is the essence of the constraint-oriented specification style [21,22,30].[3]

**Temporal decomposition** emphasises time-related phases of a system instead of functions or constraints. A system typically has distinct stages of operation, suggesting a natural, time-related decomposition of its specification. The decomposition by time may be undertaken at many levels of

[3]Work at the University of Oxford by Carroll Morgan and Tony Hoare was influential in inspiring the approach.

granularity, from coarse-grained phasing to fine-graining pipelining.

**Spatial decomposition** deals with 'space' in a topological rather than geometrical sense. The important issues are separation, distribution and replication of elements. Spatial isolation may refer equally to physical elements (hardware) and to logical elements (software). A spatial decomposition defines boundaries that may cross functional, constraint or temporal boundaries. Such a decomposition may distribute one function over several elements, may separate different parts of one view, or may combine several phases. Object-oriented analysis may be considered as a kind of spatial decomposition.

Decomposition stops when the specifier considers that the architecture has sufficiently manageable elements. This may not be an easy decision, and carries the danger of decomposing too far. A specification architecture derived in this way may be so highly structured that it has a strong influence on the implementation architecture. Over-specification must be avoided in a requirements specification, but of course a high degree of structure and detail is expected in an implementation specification.

Decomposition might be termed **architectural refinement** – the elements and their means of combination are elaborated from level to level. Architectural concerns apply throughout all levels of decomposition but the nature of the concerns changes. It is helpful to ensure that successive architectures are compatible. A discontinuity may lead to errors or may make it impossible to show that one architecture is a valid refinement of another. Figure 2 illustrates compatibility in a geometrical way. The boundaries in Architecture 1 can be found in Architecture 2a; the new architecture has compatibly decomposed the original elements. Architecture 2b is not however compatible with Architecture 1; there is a structural mismatch. Nonetheless, there are times when a major change in architectural elements is unavoidable – usually when there is a distinct change in level of abstraction, as when moving from a problem-oriented view to a solution-oriented view.

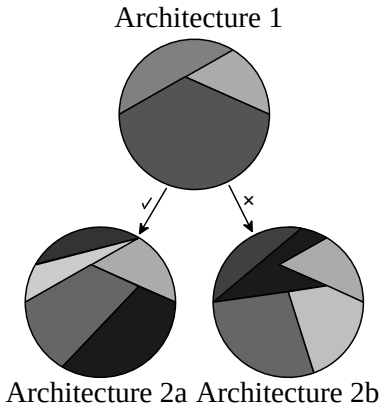



**Figure 2 Compatible and Incompatible Architectural Refinements**

In a system sense, one architecture compatibly refines another if it preserves the same interfaces while decomposing the original elements. Although elements may be decomposed in a strictly hierarchical fashion, it may nonetheless be possible to partition the functions of an interface across subsidiary components. A structural mismatch arises if an original interface disappears or is changed.

## 2.2 Examples of Modularity

A functional decomposition of the message switch might yield the following functions: message-handling (message reception, buffer management, message transmission) and routing (routing decision, internal message transfer).

A constraint decomposition of the message switch might yield constraints on the following: format and ordering of received messages; relationship between messages received and transmitted; for-

mat and ordering of transmitted messages.

A temporal decomposition of the message switch might yield the following alternative approaches: time-division multiplexed switching (time-sliced processing of each message in stages) or pipelined switching (overlapping reception, routing and transmission).

A spatial decomposition of the message switch might yield the following alternative approaches: a switching element and a control element, or replicated point-to-point switches between the message switch input and its outputs.

As an example of architectural discontinuity, consider refining the communications architecture from the functional decomposition above into an implementation-oriented architecture. In abstract terms, a computing implementation consists of transformation functions (abstract processors), communication functions (abstract buses or interfaces) and storage functions (abstract memories or storage devices).[4] The relationship between these two architectures is summarised in Table 2. The ticks in the figure indicate how problem-oriented functions could be realised by solution-oriented functions, thereby establishing the relationship between the architectures.

| | | Computing Architecture | | |
|---|---|---|---|---|
| | | Transformation Function | Communication Function | Storage Function |
| Communications Architecture | Message Reception | ✓ | ✓ | |
| | Buffer Management | ✓ | | ✓ |
| | Message Transmission | ✓ | ✓ | |
| | Routing Decision | ✓ | | ✓ |
| | Internal Message Transfer | | ✓ | |

**Table 2 Communications/Computing Architecture Relationship**

## 2.3 Criteria for Modularity

A modular architecture should exhibit coherence, decoupling and proportion. In software engineering, the concepts of cohesion and coupling are used when talking about modularity [20].

**Coherence** of an element has many aspects. The element should bring together closely related aspects – functions, constraints or whatever. Related data should be encapsulated in the same element. A coherent element will also hide internal detail such as structure that should not be visible at a higher level. At a concrete level, an element should also use closely related technologies.

**Decoupling** of elements is needed; again, software engineering practice recognises many forms of coupling. Decoupling should reflect a separation of concerns among elements. Interfaces between elements should be narrow, i.e. have a limited range of interactions and require a limited volume of information to be exchanged. Elements should act as self-standing modules, with a high degree of independence between them. Decoupling should also allow replacement of elements in case a better or different implementation should be devised in future.

**Proportion** requires a balance between breadth and depth of decomposition. A broad and shallow tree may introduce too many elements at each level for proper control and understanding. A narrow and deep tree may introduce unnecessary levels of refinement that confuse the decomposition strategy. It is often suggested that an element should be refined into no more than seven elements, though a little fewer is better. The notion of proportion is related to span of control in management science.

[4]This categorisation was evolved by the PANGLOSS project discussed in Section 5.

## 2.4 Examples of Modularity Criteria

Modularity criteria apply to any kind of decomposition strategy; as an example, only functional decomposition of the message switch will be considered in the following. The criteria are perhaps best understood when they are seen to be broken.

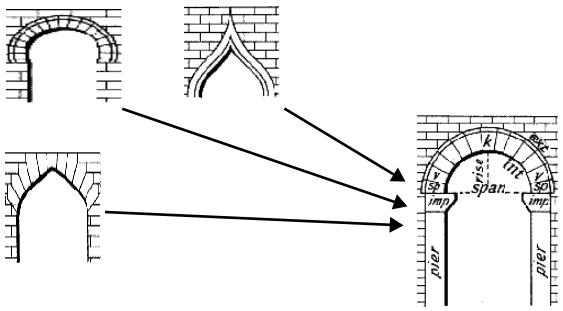
Suppose that the message switch decomposition yielded an element with the following functions: validation of input message headers, update of a routing table and buffer management. There is no strong logical relationship among the functions of this element. The functions also deal with rather different kinds of data (message header, routing table, buffer) and so do not encapsulate closely related data. Although buffer management details are hidden by this function, only parts of message handling and routing are treated. Clearly this is a not a coherent element.

Alternatively, suppose that the whole message switch were decomposed into the following elements: handling message headers, fragmenting messages, analysing message addresses and routing. Since fragmentation control is usually defined by message header fields, there would be close interaction between the elements dealing with these. Similarly, address analysis would require close interaction with header handling and with routing. Address analysis, routing and fragmentation could also be linked since the path taken by a message may depend on packet size limits along the route. Fragmentation might be affected by choice of route. Clearly this set of elements would have awkward interdependencies and so lack proper decoupling.

Functional decomposition of the switch might all be carried out at one level, leading to a set of elements such as buffer allocation, message reception, analysis of incoming messages, routing decision, routing update, internal message transfer, construction of outgoing messages, message transmission and buffer deallocation. A single-level decomposition like this would introduce a lot of complexity at one step, and would risk premature structure and omission of requirements. Conversely, a very restrained decomposition might initially lead to elements for message-handling and routing only. These might then be broken down progressively into a few elements at each level. The resulting decomposition tree could be very deep, introducing little at each level and masking the natural structure of the problem. The ideal approach would be somewhere in between, achieving proportion between breadth and depth of expansion.

# 3 Generality

## 3.1 The Principle of Generality




The principle of generality tends to be either neglected or over-applied. The architect must strike a balance between specificity and genericity. An architecture specific to current requirements may be lean and effective, but be extremely hard to modify when requirements change. A very general architecture may allow for future enhancements and functions not yet envisaged, but be too cumbersome for foreseeable needs.

Software is a peculiarly malleable product. Architects in the civil engineering sense seldom have to allow for major alterations in the future. It is hardly conceivable, for example, that a bridge would be designed to cater for moving it to another place or to serve an entirely new function [17: Chapter 1]. Nonetheless, software engineers are often expected to accommodate large changes in requirements.

**Parameterisation** is one way of making a system more general. What is given and fixed now (‘constants’, operations, types of users, etc.) may change in future. Value parameterisation as used with software procedures may be helpful here. Structural parameterisation can allow for replication of

elements. It is unwise to assume that there will always be exactly one of a certain element; new requirements for distribution, fault tolerance or parallel processing may arise. Parameterisation can become excessive and interfere with straightforward use, so sensible parameter defaults should be provided where possible.

**Generalisation** avoids *unnecessary* restrictions in behaviour, environment or technology. Of course, there will always be necessary restrictions that are an inherent and immutable part of a problem. A useful discipline is to try imagining alternative ways of structuring or realising a specification. If there are conceivable alternatives that have a major impact on the architecture, then the architecture is insufficiently general and open-ended.

**Unification** is a powerful means of ensuring generality. Similarities between elements should be identified, and the elements unified as long as this respects the need for coherence. Where elements are similar but not identical, the need for differences should be questioned.

In applying these techniques, the architect must consciously consider how the requirements might change due to new uses, changes in the system environment and developments in technology. Accurate predictions are unlikely, but it is possible to query systematically the stability of all assumptions and present requirements. A technique such as 'lateral thinking' may be helpful in breaking out of fixed patterns of thought. 'What if?' questions can shed new light on supposedly fixed issues.

## 3.2 Examples of Generality

The message switch is likely to require queues of messages waiting to be transmitted. A specific solution would be to treat queues as first-in first-out. However, many others forms of queue could be imagined. A double-ended queue, for example, would allow a message that is not transmitted correctly to be re-inserted at the head pending a retransmission attempt. Although messages may be treated equally at present, it is not hard to imagine a future extension in which messages could have degrees of urgency; a priority queue would therefore be worth considering.

The buffers of the message switch could also be specified at various levels of generality. Fixed size messages would suggest fixed size buffers. However, assuming that one message always equated to one buffer would be unduly restrictive. A wiser decision would allow buffers to be chained for larger messages. Alternatively, buffers of arbitrary size might be chosen. An even more general solution would be to allocate fixed size buffers initially, but to allow buffers to grow to the message size.

## 3.3 Criteria for Generality

Criteria from systems theory and software engineering can be applied to check for adequate generality.

**Abstractness** ensures that irrelevant details are omitted from the architecture. An abstract architecture is a model of the system, and like all models is scaled down in some way. The omitted details may concern internal interfaces, algorithms or structures. Even functional aspects may be omitted if the architecture concentrates on, say, physical aspects. Architects with an engineering background will be accustomed to placing practical considerations first; they may have to consciously practise a more abstract approach. A common way of expressing this is that abstraction deals with 'what' not 'how'. However, a balance must be sought between choosing an architecture that is overly abstract and one that is overly concrete. Over-specification leads to details that are not properly part of the requirements. Over-generalisation leads to heavy architectures that are hard to understand and to particularise for some task. It should be easy to specialise an architecture for a more limited purpose.

**Commonality** aims to absorb arbitrary differences. Re-using a library of existing elements encourages commonality, though there are problems in defining sufficiently general elements that can be instantiated easily. Object-oriented approaches tackle this kind of problem. Elements of an architecture need not be implementation elements; they might include functional specifications or constraints for example. More interestingly, a library can contain combinations as well. This is very common in engineering, where ready-made structures (e.g. circuit designs) can be exploited [25].

**Adaptability** to foreseeable changes is needed. Such changes may affect requirements directly, of course, but may be external to the system – in its environment or implementation technology. A general architecture will certainly be implementation-independent. This is particularly important in the world of computing, where each decade sees sweeping innovations in hardware and software capabilities. A useful technique in checking for implementation independence is to deliberately think of other realisations (e.g. a hardware treatment of what is currently perceived as a software problem). If such alternatives fit poorly with the current architecture then it may need rethinking.

## 3.4 Examples of Generality Criteria

The essence of the message switch is that it routes an input message to a selected output. There are many possible treatments of such a routing decision, all susceptible to considerations of generality.

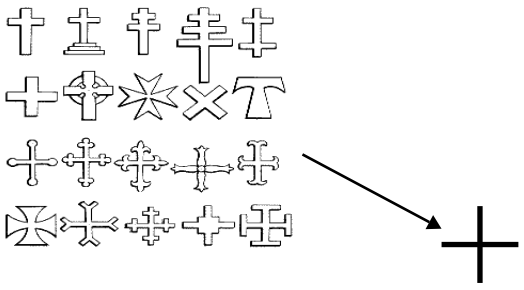
Abstractness requires that the essence of routing be captured without becoming mired in details of the routing strategy or routing update mechanism. A communications architect might immediately think of a routing *table*, but that of course is not an essential ingredient of routing. Routing might examine traffic history or be purely algorithmic, perhaps in consultation with neighbouring switches. It would be important to ensure that the routing architecture allowed sophisticated approaches such as backwards-learning, but could easily be restricted for simple approaches such as flooding.

Commonality would suggest a library of architectural elements to support routing. These would include specific routing algorithms and data structures, but could also include more abstract specification elements such as assertions to be satisfied by all routing strategies.

Adaptability would take into account factors such as new applications of routing, changes in network technology and different implementations of routing. A network routing algorithm might, for example, be adapted to message routing inside a parallel computer. Changes in network technology might introduce new factors into routing such as message priority, real-time constraints and network topology. Advances in implementation technology might allow use of storage techniques like contents-addressable memory for a routing table, or might introduce specialised routing hardware.

# 4 Simplicity

## 4.1 The Principle of Simplicity



The principle of simplicity is complementary to generality; indeed it may be in tension with it. Generality seeks a common solution that unifies differences, whereas simplicity seeks to remove differences where they are unnecessary. Simplicity leads to a more abstract architecture because irrelevant details are ignored; the architecture is not necessarily more general, however, because it may have less capability due to the elimination of features.

**Idealisation** ignores constraints until they have to be taken into account. A cleaner architecture often results from taking a ‘blue sky’ approach initially, ignoring practical restrictions. Only later need these limitations be addressed. This incremental style of architectural refinement allows early concentration on the essential features. Perhaps paradoxically, specifications are often simpler without bounds (unlimited sequence numbers, unbounded queues, arbitrarily fast transmission rate, etc.). A constraint-oriented style of specification can allow bounds to be added later as separate issues.

**Deferment** is necessary to ensure a controlled development of an architecture. Details should not be introduced too soon, and should be dealt with in a systematic way. Note that deferment considers when to introduce detail, whereas proportion concerns the balance of breadth and depth in the decomposition tree. Deferment should be exercised during architectural refinement, since there is always a

temptation to make too large a jump in abstraction level. During top-down architectural development, implementation concerns should be gradually eased into the process. Initially these concerns should be large-scale matters such as physical distribution and types of resources. Later, boundaries should be firmed up and interfaces defined. Communications interfaces, storage mechanisms, processing resources, data types and algorithms can then be addressed. Details such as the specific hardware, operating system or programming language should intrude only in the final stages.

**Minimisation** requires one solution where several are offered – the Occam's razor of the architect. It is therefore important to avoid special cases unless there is strong justification. Minimisation also requires consistency: a small set of basic constructs should be used with a fixed and unambiguous meaning.

## 4.2 Examples of Simplicity

The message switch would in practice have to cope with a variety of practical restrictions. Nonetheless, an idealised architecture could be defined with some simplifying assumptions. It might be assumed, for example, that buffers are unlimited in number, internal errors do not occur, communications links and neighbouring switches do not fail, and routing is fast enough to avoid an internal backlog.

On first decomposing the message switch, the details of routing should not appear. Only later would internal communication paths and routing strategies be defined. The switch is probably simple enough not to need deferment in considering too many details too early during refinement.

Message processing in the switch offers some examples of special cases to be avoided, at least initially. A low-level architecture might have separate buffer pools for incoming data messages, outgoing data messages, routing update messages and acknowledgements. At a higher level these should all be treated simply as message buffers. Similarly, a low-level architecture might prescribe special processing for routing messages and management messages to be consumed by the switch itself, but at a higher level this would be a premature consideration.

A good example of one thing having several meanings is the Disconnect Request message used in many protocols. This can mean 'connection attempt refused' when used during connection establishment, and 'connection released' or 'connection failed' when used during data transfer. Half-open connections lead to ambiguity of this signal. Ideally, there should be separate disconnection messages. Note that simplicity here would avoid ambiguity in concepts by increasing their number! Having too few concepts and overloading them inconsistently does not simplify matters.

## 4.3 Criteria for Simplicity

The criteria for satisfactory application of simplicity to some extent mirror those for generality.

**Uniformity** requires unnecessary differences to be eliminated. Symmetry is one aspect of this. Care should be taken over introducing asymmetrical treatment such as a general function that only certain elements can invoke, the presence of some but not all complementary functions, or universal rules that are suspended in some circumstances.

**Elegance** is subjective and hard to define; inelegance is perhaps easier to recognise. Examples of inelegance include *ad hoc* constructions, complex interdependencies, and large or monolithic architectures. Elegance is found in compact architectures that clearly embody the essence of a problem.

**Economy** should show in the use of few but powerful constructs. Restricting the number of architectural elements and their types of combination promotes economy. There should also be economy in the distribution of features. Ideally, features should be localised where possible (where locality means logical not physical boundaries).

## 4.4 Examples of Simplicity Criteria

Uniformity in buffering messages inside the switch would ignore their origin, destination, type and

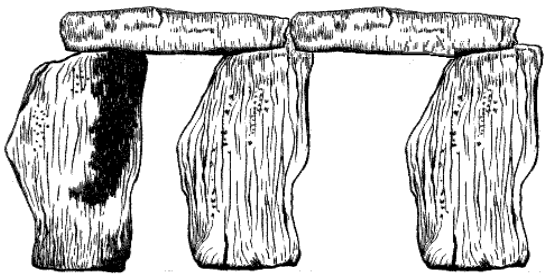
size. Although these factors might be grounds for separate treatment in the actual implementation, this would be at the expense of simplicity. Uniformity would also require that any switch be able to route via any other, though there may be cost or quality reasons against this for particular messages. It might be considered that checksum processing should be undertaken only for external messages. However, a more symmetrical view would extend this to internal messages as well if it were necessary to ensure the same level of protection. The switch may choose only to fragment messages when required by the next hop, whereas the complementary function of internal reassembly might be desirable. Always allocating buffers to messages would achieve consistency. In practice, not using buffers for acknowledgements might be desirable since acknowledgements are generally short and fixed-length. However, special-case processing of acknowledgements would add new complexity.

The factors involved in routing decisions are many and varied: cost, throughput, security, reliability, fragmentation limits, etc. An elegant and economical treatment might reduce all factors to single numerical quantities which had to be minimised, taking the weight of each factor into account. If the architecture treated the entire processing of a message as a single activity, it would be monolithic and lacking in manageable structure. Special-purpose processing of particular kinds of messages in particular circumstances would be inelegant. The switch might have to treat acknowledgements specially under buffer overload, for example, because they free buffers for outgoing messages. Such complexity may be justified in the implementation, but should not intrude in early stages of the architecture.

Economy would require a limited number of data types in the switch (e.g. message content, message address) and a limited number of operations on them (e.g. receive and transmit, route). Localisation of buffer management would imply that buffers were allocated and deallocated by a single element inside the switch.

# 5 Architectural Case Study: A High-Speed Message Switch

## 5.1 The PANGLOSS Project



The architecture of a high-speed message switch was developed by the PANGLOSS project (Parallel Architecture for Networking Gateways Linking OSI Systems, ESPRIT 890 [1, 24]). Within the project, work on developing the switch architecture was undertaken by an Architecture Task including the author. Internetworking was required to follow the principles of OSI.

The expectations of the message switch were demanding. It had to support a variety of scenarios and traffic patterns: connection-less, connection-oriented and stream-oriented. Throughput per subnetwork attachment point was planned at 2700 packets per second, each packet being 128 bytes (i.e. about 2.8 Mbits per second). Staging delay (the delay caused by the switch) was felt to be important in a high-speed switch; 95% of messages had to be dealt with in less than 30 milliseconds. To allow performance of the switch to be scaled up, the switch had to allow extra processing capacity to be added. Performance was to be extensible up to 45 times the base capacity, with at most a 25% deviation from linearity in performance vs. added capacity.

The use of formal approaches was felt to be essential as the switch was a complex and concurrent system. As the specification architecture was developed, it was formalised in LOTOS. This gave precision to the architecture, and allowed rigorous analysis of the design for functionality and performance. A series of architectures was evolved, beginning with the basic communications requirements, proceeding to a reference architecture for the switch, and ending up with a hardware architecture. The hardware architecture was chosen to be a collection of transputers linked by high-speed interconnection structures. A strategy was developed for translation of low-level LOTOS into *occam* [15], the prin-

cipal language of the transputer. Performance analyses were based on both communications architectures and hardware architectures.

The architectural approach presented in this paper was not explicitly used during PANGLOSS. However, the evolution of the PANGLOSS architecture is a good example of how the approach can be applied. The following subsections illustrate only a few of the principles and techniques at work.

## 5.2 Internetworking Architecture

In conformance with OSI, the internetworking architecture of the switch was based on the Internal Organisation of the Network Layer [10]. As shown in Figure 3, this decomposes the network layer into three sublayers:

**3C (Internet)** This sublayer presents a uniform network service and uniform network addressing to all end systems. It also handles routing across subnetworks and deals with congestion control at the level of the global network. Since the message switch acts purely as a relay, there are no 3C service access points. In practice these might be necessary to handle routing or management messages destined for the switch.

**3B (Subnetwork Convergence)** This sublayer harmonises differences between the global network service and the individual service offered by one subnetwork. This may require enhancing less comprehensive subnetworks or even hiding unneeded services in more sophisticated subnetworks.

**3A (Subnetwork Access)** This sublayer provides the service offered by just one subnetwork.

Layer 2 (Data Link) deals with reliable transfer over one transmission link, while layer 1 (Physical) deals with low-level aspects. 3C is common across the global network, but sublayers from 3B downwards are particular to a subnetwork. Although there may be many subnetworks attached to the switch, only two will be shown in subsequent figures for simplicity. The central 'pillar' in Figure 3 is shown shaded to suggest another subnetwork. Service access points are shown as ovals in the figure.

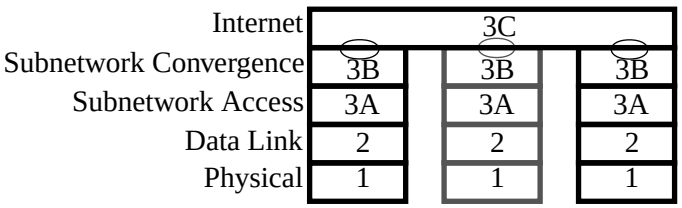



**Figure 3 (Sub)layering within the Internetworking Architecture**

The PANGLOSS project used the internetworking architecture to develop the switch architecture. The following description shows how some of the principles in this paper can (retrospectively) justify the switch architecture.

**Modularity (Functional Decomposition)** The initial functional decomposition of the switch in Figure 3 was determined by the OSI architecture.

**Simplicity (Deferment)** It is assumed that the 3A, 2 and 1 services will be realised later in the design by standard subnetwork modules. This is a reasonable assumption because equipment suppliers will normally provide drivers for major subnetwork types, so the provision of these services can be deferred.

**Simplicity (Deferment)** The 3B service matches the standardised 3C service by definition of the network sublayers. The mapping between the 3B service and the various 3A services is then entirely dependent on subnetwork type. Again it is reasonable to conclude that equipment

suppliers will provide an adaptor to offer the global network service over major subnetwork types. The conclusion from this and the previous simplification is that the 3C sublayer is the key feature of the architecture. It is therefore possible to concentrate on the internetworking aspects that form the essence of the switch.

**Simplicity (Minimisation)** 3B service access points are interconnected via the 3C sublayer in exactly the same way as the service access points of a subnetwork. In other words, the switch itself contains a subnetwork![5] The architecture for OSI networking can therefore be applied recursively within the switch rather than develop a new architecture.

**Modularity (Functional Decomposition), Simplicity (Minimisation)** 3C can be decomposed into Partial Internet Functions (PIFs) and an internal subnetwork. Following OSI principles, the internal subnetwork must offer an internal 3B service. The PIFs together form the original 3C function. Each PIF entity need connect only one external 3B service access point to one internal one as shown in Figure 4. A one-to-one relationship between service access points is simplest, but can realise all the required intercommunication.

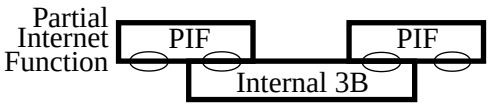



**Figure 4 Decomposition of the 3C Sublayer**

**Generality (Unification)** The internal 3B must carry different kinds of traffic, all of which should use a common core of communication. At this level of abstraction it would be inappropriate to define different kinds of internal subnetwork for different kinds of traffic.

**Modularity (Functional Decomposition)** Just as with the OSI internetworking architecture, the internal subnetwork should be decomposed into a common Interconnection Function (IF, the raw internal subnetwork) and a Harmonised Interconnection Function (HIF, to adapt the raw internal subnetwork to the 3B service). As with any OSI subnetwork, the service offered by the IF may not match the global network service, so a convergence function (the HIF) is necessary. The result is shown in Figure 5.

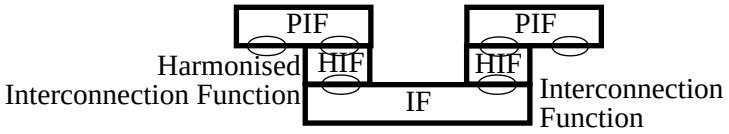



**Figure 5 Decomposition of the Internal 3B**

## 5.3 Internal Architecture

Now that the basic decomposition of the switch has been determined, attention can focus on more detailed architectural issues. In the following, note that significant progress can be made at an architectural level without having to consider implementation matters in detail. However, general implementation concerns now begin to influence the architecture. In particular, efficiency issues are addressed at an architectural level. Observe that a common pattern in developing the architecture is first to consider general approaches and then to simplify them to a specific solution. References to 'transmitting/receiving HIF' in the following mean the HIF instance that is transmitting/receiving data

[5]Though, of course, the internal subnetwork is not visible in the OSI sense.

via the IF to/from its peer.

**Generality (Generalisation)** Internal message transfer may use message passing or pointer passing. Message passing is a clean solution but may have undesirable performance penalties. Message copying is time-consuming and inefficient. Messages could be large, leading to long copying delays. The additional complication of internal segmentation would therefore be advisable to deal with size problems and to reduce delays. On the other hand, pointer passing is efficiently realised through shared memory. In a closely coupled system such as the switch, the requirement for shared memory could be easily satisfied. Pointer passing improves high-speed processing of message streams, avoiding delays while buffers are being copied.

**Simplicity (Minimisation)** Message passing is most appropriate for transfer across a communications medium, and therefore at the HIF↔IF interface. Pointer passing is most appropriate for transfer between closely coupled systems, and therefore at the PIF↔HIF interface.

**Generality (Parameterisation)** The PIF and HIF entities should be replicated per subnetwork attachment since there may be connections to many subnetworks.

**Simplicity (Minimisation)** The IF is unique in the switch, and therefore should not be replicated unless it is strictly necessary. This might, however, lead to the IF becoming the main bottleneck in the switch. Internal flow control would then be necessary in order to allocate switch capacity fairly and to prevent heavy traffic streams from interfering with others. If this can be satisfactorily addressed then a single IF should be used, otherwise the IF should be replicated.

**Generality (Generalisation)** Flow control might be based on knowing the capacity of an IF stream or a receiving HIF entity in advance. However, this is unrealistic due to the bursty nature of much communications traffic. Other flow control mechanisms might include using a window, credit, choke packets or pacing transmissions to match acknowledgements from the receiving HIF. Unfortunately these all add undesirable complexity to the IF protocol, and it is already clear that performance of the IF should be optimised.

**Simplicity (Minimisation)** The best solution is for the transmitting HIF instance to negotiate in advance the capacity needed from the IF and the receiving HIF instance. A HIF instance is connected through its local PIF instance to a particular subnetwork and so has the information required to determine this capacity.

**Generality (Generalisation)** Internal buffering is best avoided where possible since it makes shared memory a bottleneck. Buffer management strategies could become complex, and would consume processor power and increase delays. In general, buffering is needed by a transmitter to retransmit messages or to cope with delays while waiting to transmit. In general, buffering is needed by a receiver to resequence misordered messages.

**Simplicity (Minimisation)** As a small-scale, integrated subnetwork the IF is likely to have a very low error rate so retransmission should not be an issue. Resequencing messages should also not be an issue in the IF. The only justifiable need for buffers is therefore to hold messages until they can be transmitted internally. It follows that buffering should be used on the transmitting HIF→IF interface. Buffering might similarly be considered for the receiving PIF→3B interface, but its use could lead to problems. Since there is no direct flow control over the IF, transmitting data from a high-speed subnetwork to a low-speed one could saturate the IF. The correct place for buffering is therefore in the transmitting HIF instance for the HIF→IF interface.

**Generality (Parameterisation)** In order to scale up the capacity of the switch, it must be possible to replicate certain of its elements. In general, replicated processing is appropriate where independent processing of messages is logically possible and when parallel processing is desirable. For the HIF, this means that each pair of transmitting/receiving HIF instances should be replicated. Message queues should then be introduced for each transmitting HIF instance,

avoiding one destination affecting traffic to other destinations served by that instance.

# 6 Conclusion

The importance and role of specification architecture have been explained. It has been argued that specification architecture not specification writing is difficult. As summarised in Table 1, three main principles and associated techniques and criteria have been proposed for developing effective architectures: modularity, generality and simplicity. The approach is general and could be used with many classes of system. However, it has been illustrated in the context of communications with the example of a message switch. The PANGLOSS example of a high-speed message switch also provided a larger and more realistic case study.

The list of principles for ensuring a clean and effective architecture could be extended much further. It could be worthwhile to compare the approach to that of 'patterns' and 'frameworks' in the object-oriented community. It is hoped that the brief presentation in this paper will encourage interest in architectural matters, and will bring specification architecture into proper focus.

# Acknowledgements

This paper is derived from an unpublished tutorial given by the author at FORTE IV. The conference organisers, Overseas Telecommunications Corporation[6] and the University of Queensland, generously supported the author's attendance to present the tutorial. The PANGLOSS case study reflects the architectural work of many people on the project, particularly from the University of Twente, the University of Liège and CAP.[7] PANGLOSS was supported by the European Commission under the ESPRIT programme. Architectural images in the paper were adapted from *Digital Webster*, the on-line version of [31] supported by NeXT Inc. under the NEXTSTEP™ environment. Luigi Logrippo (University of Ottawa) made many insightful observations on a draft of the paper. Helpful comments were also received from Marten van Sinderen (University of Twente) and Richard Sinnott (University of Stirling). The paper benefited from a careful reading by the anonymous reviewers.

# References

[1] Kees Bogaards. LOTOS supported system development. In Kenneth J. Turner, editor, *Proc. Formal Description Techniques I*, pages 279–294. North-Holland, Amsterdam, Netherlands, 1989.

[2] Kees Bogaards. *A Methodology for the Architectural Design of Open Distributed Systems*. PhD thesis, University of Twente, Enschede, Netherlands, 1990.

[3] Howard Bowman, John Derrick, Peter Linington, and Maarten W. A. Steen. Cross-viewpoint consistency in Open Distributed Processing. *Software Engineering Journal*, pages 44–57, January 1996.

[4] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):153–159, 1992.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Addison-Wesley, Reading, Massachusetts, USA, 1994.

[6] Now Telecom Australia.

[7] Now part of the SEMA group.

[6] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.

[7] Reinhard Gotzhein. The formal definition of architectural concepts: 'Interaction points'. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*. North-Holland, Amsterdam, Netherlands, December 1989.

[8] Reinhard Gotzhein. Formal definition and representation of interaction points. *Computer Networks and ISDN Systems*, 25(1):3–22, August 1992.

[9] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*. ISO/IEC 7498. International Organization for Standardization, Geneva, Switzerland, 1984.

[10] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Internal Organization of the Network Layer.* ISO/IEC 8648. International Organization for Standardization, Geneva, Switzerland, 1987.

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

[12] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Guidelines for the Application of ESTELLE, LOTOS and SDL*. ISO/IEC TR 10167. International Organization for Standardization, Geneva, Switzerland, 1990.

[13] ISO/IEC. *Open Distributed Processing – Basic Reference Model – Part 4: Architectural Semantics*. ISO/IEC 10746-4. International Organization for Standardization, Geneva, Switzerland, August 1994. ISO/IEC JTC1/SC21/N9035 and N9036.

[14] ISO/IEC. *Information Processing Systems – Open Distributed Processing – Basic Reference Model*. ISO/IEC 10746. International Organization for Standardization, Geneva, Switzerland, 1995.

[15] Inmos Ltd. *occam 2 Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[17] Stephen R. Schach. *Software Engineering*. Aksen Associates (Irwin), Homewood, Illinois, USA, Second edition, 1993.

[18] Jeroen Schot. *The Role of Architectural Semantics in the Formal Approach to Distributed System Design*. PhD thesis, Department of Informatics, University of Twente, Enschede, Netherlands, 1992.

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

[20] W. Stevens. *Software Design: Concepts and Methods*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1990.

[21] Alastair J. Tocher. LOTOS and the formal specification of communications standards: An example. In Peter N. Scharbach, editor, *Formal Methods – Theory and Practice*, pages 5–51. BSP Professional Books, Oxford, UK, 1989.

[22] Kenneth J. Turner. The alternating bit protocol — Constraint-oriented specifications in LOTOS. In *Proc. British Computer Society Workshop on Formal Methods in Standards*, pages 1–13, London, April 1988. British Computer Society.

[23] Kenneth J. Turner. An architectural semantics for LOTOS. In Harry Rudin and Colin H. West, editors, *Proc. Protocol Specification, Testing and Verification VII*, pages 15–28. North-Holland, Amsterdam, Netherlands, October 1988.

[24] Kenneth J. Turner. A LOTOS-based development strategy. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 157–174. North-Holland, Amsterdam, Netherlands, 1990.

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

[26] Jeroen van de Lagemaat, Tommaso Bolognesi, and Chris A. Vissers, editors. *The LOTOSPHERE Project*. Kluwer, 1995.

[27] Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz, editors. *The Formal Description Technique LOTOS: Results of the ESPRIT SEDOS Project*. Elsevier Science Publishers, 1989.

[28] A. J. M. van Gasteren. *On the Shape of Mathematical Arguments*, volume 445 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1990.

[29] Andrew J. Vickers and John A. McDermid. An approach to the design of software for distributed real-time systems. Technical Report YCS 211, Department of Computer Science, University of York, UK, October 1993.

[30] Chris A. Vissers, Giuseppe Scollo, and Marten van Sinderen. Architecture and specification style in formal descriptions of distributed systems. *Theoretical Computer Science*, 89:179–206, 1991.

[31] Webster. *Ninth New Collegiate Dictionary.* Merriam-Webster, New York, 1988.