Kenneth J. Turner. Relating architecture and specification (pre-publication version). Computer Networks and ISDN Systems, 29(4):437-456, March 1997.

Relating Architecture and Specification

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 problems of multiple specification languages for multiple architectures are discussed. It is concluded that an architectural semantics is of practical value in ensuring consistent and effective development of specifications. The approach is discussed in general and by examples, concentrating mainly on an architectural semantics for Open Systems Interconnection in relation to ESTELLE, LOTOS and SDL. It is shown how an architectural semantics can be realised using a library of specification templates, allowing the specifier to work at a higher, more architectural level. Some LOTOS templates are discussed, mainly for Open Systems Interconnection.

Keywords: architecture, architectural semantics, FDT (Formal Description Technique), LOTOS (Language Of Temporal Ordering Specification), specification, specification architecture, specification template

1 Introduction

1.1 The Problem of Multiple Languages

What language might be used to specify problems in a given domain? A number of languages have been developed for specific kinds of application. For example, in the field of communications systems there are the standardised FDTs (Formal Description Techniques). These are ESTELLE (Extended Finite State Machine Language [9]), LOTOS (Language Of Temporal Ordering Specification [10]) and SDL (Specification and Description Language [18]). Other standardised techniques for communications systems offer a rigorous, if not (mathematically) formal, approach: ASN.1 (Abstract Syntax Notation 1 [12, 13]) and TTCN (Tree and Tabular Combined Notation [14]). In the field of hardware description there are many widely used languages such as CIRCAL (Circuit Calculus [23]), HOL (Higher Order Logic [3]) and VHDL (VLSI Hardware Description Language [7]). Although the languages just cited are tied by origin to a particular domain, they might be considered general-purpose languages that could be used on a wide variety of problems.

In the computer science community generally, an even wider range of non-standardised specification languages is also available. Languages reaching standardisation or in widespread use include

CCS (Calculus of Communicating Systems [24]), CSP (Communicating Sequential Processes [6]), VDM (Vienna Development Method [19]) and Z [28]. Again these are languages with broad applicability.

The rich choice of specification languages is good because it allows an appropriate selection for the kind of problem to be specified. However, there could be difficulties when comparing specifications of the same problem in different languages. Similarly there could be difficulties when combining specifications in different languages of parts of the same system. Such difficulties are particularly acute in standardisation, where there is strong interest in specification but multiple incompatible specifications are unacceptable. (Unfortunately, deciding whether two specifications are compatible may be hard – or even impossible if they are in different languages.) Even if only one specification language were used, stylistic differences between specifiers could make it hard to relate specifications of the same system or systems from the same class. Of course, it is not so common that independent specifications of the same thing are written.[1] More likely, specifications in the same language may be written of subsystems that have to be combined.

A real-life example is found in the specifications written of standards from OSI (Open Systems Interconnection [8]). There it is necessary to combine specifications of adjacent layers. How can this be achieved without a common understanding by specifiers of the architectural concepts? Such concepts are typically expressed in natural language and so allow room for interpretation. When the possibility of multiple specification languages is considered, the scope for differing formalisations could be rather wide. A key issue is therefore to ensure that the concepts of a given problem domain are specified consistently, whether using one language or several.

1.2 The Problem of Multiple Architectures

Specification languages have often been developed initially for a particular class of system (e.g. OSI systems). Inevitably the designers of the language will wish to capitalise on their investment by applying the language to new kinds of system (e.g. telecommunications services). This is especially true of standardised languages since the cost of development is high and should be repaid by widespread use. However, it is rarely possible to use a language on a new class of system without considerable investigation of how best to specify the concepts of such systems. In other words it is necessary to establish an effective relationship between the concepts of the new class of system to be modelled and those of the specification language. It would be time-consuming to develop this relationship afresh for every new architecture and for every language that might be applied to it.

In general it could be necessary to consider a full mapping between all architectures and specification languages of interest – a costly solution. A practical strategy is therefore required to reduce the overheads of this approach. Relating an architecture to a specification language determines an **architectural semantics.**[2] This reflects the concepts of an architecture in a specification language. As will be seen, an architectural semantics can be developed in a way that efficiently allows for multiple architectures and multiple languages.

1.3 The Need for Architectural Specification

A specifier who worked purely at the level of the specification language would be constantly making decisions about how best to model architectural concepts in the language. This would be like an electronics engineer always having to design circuits from first principles; it could be done, but would not be cost-effective. Of course, circuits are designed in the knowledge of higher-level combinations of components: potential dividers, tuned circuits, logic gates, amplifiers, etc. Many such macro components are available directly to the circuit designer. And even if not, the designer can proceed at the level of blocks rather than elementary components, secure in the knowledge that the design of these

[1] Except perhaps for the Alternating Bit Protocol and the like!

[2] The term is due to Chris Vissers (University of Twente).

blocks is already worked out. Awareness of higher-level structures allows an engineer to look at components in a circuit diagram and recognise the blocks and their relationships. Understanding as well as design are thus enhanced by attention to architecture.

The specifier should ideally be able to work at the level of architectural components and their combinations. A certain class of feature in the system to be specified should immediately suggest a particular way of specifying it. Conversely, it should be obvious from the specification that certain parts of it correspond to well-known architectural features. This approach might be termed architectural specification. The term **specification architecture** will be used to mean the structuring of specifications, not the specification of architecture in general. An effective specification architecture can be supported by **specification templates** based on analysing the architecture of some class of systems.

Templates are the practical goal of the work, since they considerably increase the efficiency and consistency of specification. As will be seen, a template library can be supported by tools, helping to automate the process of specification – though of course it remains a highly creative task. In order to reach the goal of specification templates, several intermediate stages will be described in the paper. General consideration of specifying an architecture in some formal language will suggest an approach to defining architectural semantics. This will benefit from defining a consistent hierarchy of architectural concepts. Many of these can be mapped onto more fundamental concepts, which will also be developed as a consistent hierarchy. The fundamental concepts are relevant because they are common to a number of architectures. Formal representations can then be defined for the chosen specification language; specific examples in LOTOS will be given. The taxonomy of architectural concepts will be useful here in achieving a consistent formalisation. Finally, the formal representations can be embedded in specification templates that can be called up from a library.

Architectural matters are essentially intuitive and informal. To this extent it is not possible to give an entirely rigorous way of formalising architectural concepts. (In software engineering, analysis and specification similarly have to deal with informal requirements; true rigour is not possible until a formal specification of requirements exists.) However, it is certainly possible to adopt a systematic approach to formalising architectural concepts, and this is a key aspect of the approach in the paper. Architectural semantics involves two kinds of activity: conceptual analysis based on the informal description of an architecture, and specification based on a formal language. The conceptual analysis provides a framework within which consistent formalisation is possible. Of course, the architectural semantics is unlikely to be unique; there may be a number of reasonable ways of analysing the concepts of an architecture and formalising them. The combined experience of architects and specifiers will be necessary in order to choose a good approach. (Similarly, there is no one way to write a program that implements some requirements, but an experienced programmer will be knowledgeable about appropriate approaches.) The existence of *an* architectural semantics (even if it is not unique) is a powerful aid in ensuring consistency of specification.

1.4 Historical Development

An interest in specification architecture was evident from the early days of developing ESTELLE and LOTOS. Since these FDTs had to be suitable for specifying OSI at least, it was important that they reflected OSI architectural concepts in a natural way. The FDT group in ISO (International Organization for Standardization) was part of the OSI group (SC21/WG1). The FDT group was split into an FDT architecture subgroup as well as subgroups for ESTELLE and LOTOS. The architecture subgroup (which the author chaired for some years) was charged with developing a proper relationship between OSI and the FDTs. [30] reports early work on an architectural semantics for OSI using LOTOS.

The work of the architecture subgroup led to the establishment of ISO project 21.44 concerned with architectural semantics for FDTs. This project was focused on OSI, but it turned out that much of the work was directly relevant to the emerging definition of ODP (Open Distributed Processing [17]). At the same time, much of the FDT development work in the OSI area was running down. The architectural semantics project was therefore transferred to the ODP group and expanded to cover its requirements.[3] Other specification languages such as RAISE (Rigorous Approach to Industrial Soft-

ware Engineering [25]) and variants of Z have been subsequently investigated for their suitability in modelling ODP concepts.

About the same time as the OSI architectural semantics project, a parallel and related project was set up between ISO and CCITT (International Consultative Committee on Telegraphy and Telephony, now ITU-T (International Telecommunications Union)). With three standardised FDTs available, it was considered important to offer guidance to specifiers using these techniques. ISO project 21.45 (CCITT Question 6/X) dealt with guidelines for the application of FDTs [11]. Naturally there were questions of how best to model architectural concepts in each FDT. Much of the OSI architectural semantics work was therefore absorbed into the FDT guidelines.

The work on specification architecture in standards circles stimulated research elsewhere. For example, the PANGLOSS project (Parallel Architecture for Networking Gateways Linking OSI Systems, ESPRIT 890) followed an architectural approach in the formal specification of a high-speed network gateway. [1] describes how LOTOS was used to specify and design the gateway. The same project inspired an architectural approach to specification and design of communications systems [2]. An architectural semantics for distributed systems was subsequently developed [26]. Concern that specific FDTs imposed individual interpretations of architectural concepts prompted a study using temporal logic. Initially the properties of interaction points were investigated in [4], leading to wider consideration of architectural issues [5].

1.5 Paper Structure

Section 2 explains the motivation and approach behind defining an architectural semantics. As an illustration, a partial architectural semantics is worked out for OSI in relation to ESTELLE, LOTOS and SDL. Examples are also given from other fields. Section 3 shows how an architectural semantics can be realised as a library of specification templates that allow the specifier to work at a higher, architectural level. Some LOTOS templates, mainly for OSI, are used to illustrate the approach.

2 Architectural Semantics

2.1 Motivation

When formal specifications of OSI standards were first written, it soon became clear that there were many possibilities for representing architectural concepts in a formal way. To some extent this reflected the intention to keep the architecture at a broad, implementation-independent level. The varieties of interpretation also stemmed from the use of natural language in the defining documents. Having to specify broadly defined concepts in an FDT forced attention on how these concepts should be interpreted. The following examples are a few typical problems taken from the work on formalising OSI standards.

A service primitive is a key feature in OSI since it models the interactions at a service, i.e. between adjacent protocol entities. Yet a service primitive in not defined in the main OSI reference document [8]; it is defined in the OSI service conventions [15]. Even there it is not said whether service primitives are atomic, instantaneous or synchronous. Some specifiers considered service primitive occurrences to be like procedure calls. Others thought of them as asynchronous requests that were queued until they could be processed (the ESTELLE and SDL view). Yet others treated them as synchronous events (the LOTOS view). This was not mere hair-splitting, because the consequences of each view led to different behaviour being specified.

Another undefined aspect of service primitives was whether it was necessary to specify incorrect behaviour by the service user, e.g. issuing a service primitive at an invalid time. Again the specifiers

[3]The standardisation community felt that an OSI architectural semantics was less important by this time, so the OSI aspect of the project was not progressed much further.

came up with different interpretations. Incorrect behaviour could be explicitly specified but ignored (the ESTELLE view). Incorrect behaviour might be explicitly allowed but implicitly ignored (the SDL view). Incorrect behaviour might even be omitted completely from the specification and thus be beyond its scope (the LOTOS view, reflecting a preference for greater abstraction).

Service data units were also open to interpretation. Are they atomic? OSI allows for the possibility of interface data units to transfer fragments of a service data unit across a service. However, it is not clear whether a protocol entity can legitimately deal in isolation with such parts of a service data unit. It might be desirable, for example, to send protocol data units before all the interface data units have been completely transferred across the service. Interface data units might even be regarded as an implementation matter and thus not at the same level of abstraction as service data units. There is thus again a choice of how an architectural concept might be interpreted.

Several views emerged as to the true meaning of a service access point. It might be treated as a purely structural concept, representing an abstract interface between two protocol entities. However, a service access point might also be an active agent; in particular, some specifiers thought of it having a dynamic aspect to establish connections through it. In another treatment, service access points were considered to be processes that could be decomposed further into subsidiary entities. Other questions about service access points also arose. Could connection-less and connection-mode services be supported at the same service access point? Were endpoints within service access points necessarily associated with connections, or were they a more general concept?

It was clear from the experience of trying to formalise OSI standards that much more guidance was needed on interpreting architectural concepts.

2.2 Approach

An architectural semantics is intended to guide the specification of standards for a particular architecture. The aim is to interpret the concepts of that architecture in a particular specification language. Ideally such a semantics should be given denotationally, i.e. should prescribe how each architectural concept is represented in the language. However, to be prescriptive requires agreement on the 'best' approach to specification. Often there are several reasonable specification styles for a concept, and it takes time and experience to evolve a recommended practice. Even then, it may be easier to give a sample specification than to give rules for modelling a concept. Architectural semantics has therefore tended to be a mixture of the prescriptive and the descriptive.

In considering the concepts of an architecture or the constructs of a language it may be useful to distinguish between components and combinators. The components are the building blocks, and may be elementary or composite. The combinators are the means of creating more complex components from simpler ones. For an architecture, there may not be explicit recognition of combinators as such. For a language, the combinators are often called its operators.

As examples of these, consider concatenation (an OSI term) and behaviour (a language term). The concatenation combinator takes a list of protocol data units as components and builds a service data unit as component. The partial ordering combinator takes a set of actions as components and builds a behaviour as component. Since the result of combination is a component, the combinator itself may be loosely referred to as if it were the component (e.g. a concatenation of protocol data units or a partial ordering of actions). The distinction between component and combinator may therefore not be a sharp one. Examples of particular components and combinators are discussed in later subsections.

A specific instance of some architecture (e.g. an OSI protocol standard) conforms to that architecture, making reference to and being built from its components and combinators. A specific formal specification (e.g. in LOTOS) conforms to the constructs of the language, being defined by its semantics. The relationships among these elements are shown in Figure 1. If the architectural concepts have a defined denotation in terms of the language constructs, this induces a relation between the architecture instance and its formal specification. Alternatively, if the architecture instance has a defined

denotation directly in terms of the formal specification, this induces a relation between the architectural concepts and the basic language constructs. Figure 1 thus shows a kind of homomorphism.[4]

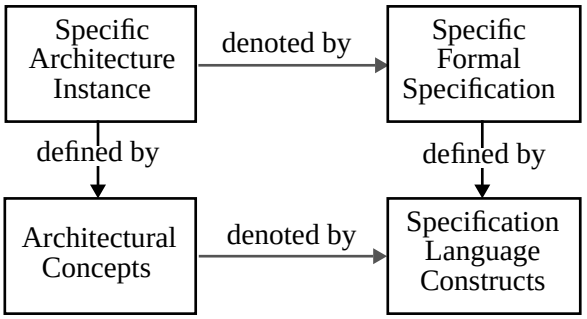

Figure 1 Relationship between Architecture and Specification

There is a choice of whether an architectural semantics is developed primarily at the level of specific architecture instances or at a more basic level. A direct mapping between all A architectures of interest and all S specification languages would require $A \times S$ mappings. This would be costly to define. However, there is a solution if the architectures can be restricted to some very general class such as information processing systems (of which OSI and ODP systems are examples). Such systems share many components out of which more complex components are built. Such systems may also share combinators for building these higher-level components. If these fundamental components and combinators can be established for each architecture and can be denoted in each specification language, then the number of mappings to be considered falls to $A + S$.

A separate issue is whether architectural concepts should be directly mapped to specification constructs or via some intermediate hierarchy that exploits the dependencies between concepts in the architecture and constructs in the language. The two approaches are depicted in Figure 2. The direct mapping in (a) gives each architectural concept a specific denotation in some language. As a result, the mapping could be complex to define, would not show a clear hierarchical relationship, and could risk inconsistent specification of concepts at different levels in the architecture. A direct mapping would also be required for each architecture-language pair. The indirect mapping is shown in (b). Here there is a clear hierarchy, and the extent of the mapping is reduced because it can be restricted to fundamental concepts.

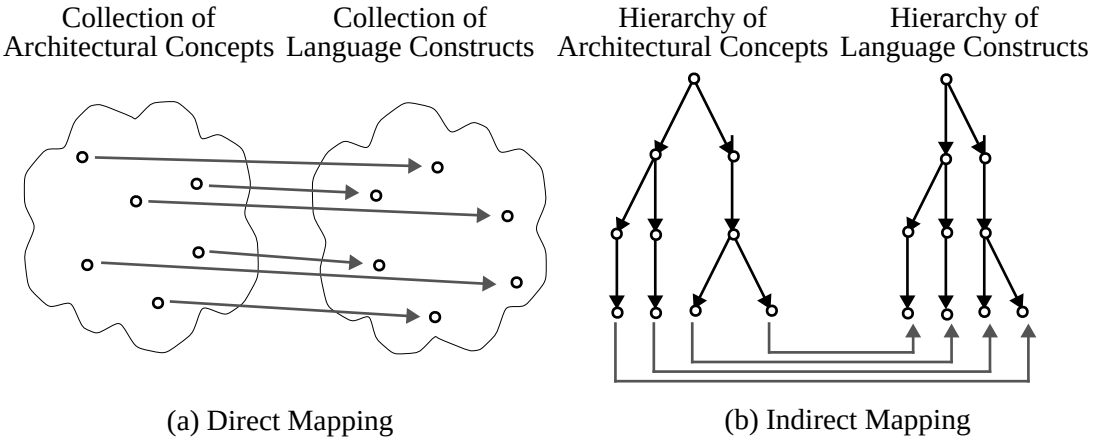

Figure 2 Relationship between Architectural Concepts and Language Constructs

[4]Or perhaps an ISOmorphism in the case of an international standard!

With many architectures and many languages, an indirect mapping between each architecture and language would be required at the fundamental level. However, an indirect mapping would result in a less obvious relationship between composite architectural concepts and their denotations. Such a mapping might offer little practical guidance to the specifier.

A compromise is thus desirable. A hierarchy of concepts and constructs should be established along with a mapping at various levels. The mapping should exploit the hierarchy and thus define the denotation of a composite architectural concept using the denotations of its constituent parts. Such a mapping would be intermediate in size and complexity between the extremes of $A \times S$ and $A + S$. In the context of some example architectures and specification languages, the mapping might be as shown in Figure 3. The $A + S$ darker arrows represent the indirect fundamental mappings, the $A \times S$ lighter arrows represent the direct architectural mappings. The formal representations obtained by fol-

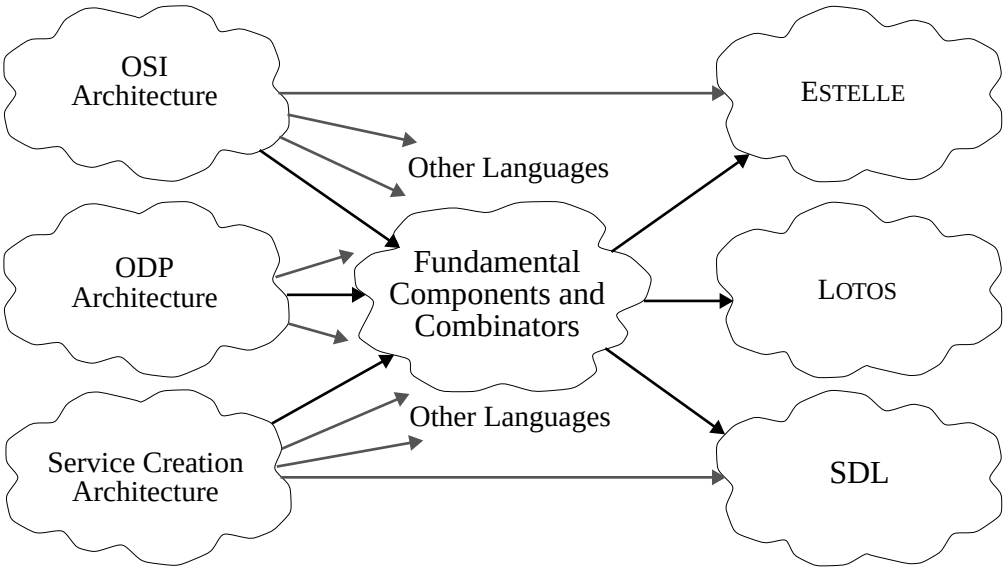

Figure 3 Mapping between Multiple Architectures and Specification Languages

lowing the lighter arrows would be preferable if the result were more compact and more comprehensible than the result of indirectly composing lower-level formal representations.

Developing an architectural semantics begins with consideration of a new architecture. A core set of fundamental information processing components and combinators has to be developed initially, and may have to be extended for a new architecture. The concepts of the architecture will usually, but not necessarily, have been defined already in a hierarchical fashion. When there is no hierarchy, part of defining the architectural semantics includes establishing this. Even if there is an existing hierarchy, it may not be sufficiently precise to allow immediate formalisation. It is unlikely that the architecture will have been expressed in terms of fundamental components and combinators; doing this is another part of the architectural semantics. Even without considering issues of formalisation, the development of a consistent concept hierarchy can be a valuable exercise.

The architectural semantics can now deal with the specification languages of interest. For a new specification language it is necessary to define a denotation for each fundamental component and combinator. If the language is insufficiently expressive this may prove to be awkward or even impossible, thus limiting the use of the language for some architectures. The final step is to provide a denotation for each composite architectural concept, building on the denotations already defined.

As will be seen from the examples about to be given, the development of an architectural semantics is relatively intricate. However, analysing an architecture is desirable to achieve a systematic formal representation of concepts

2.3 Fundamental Information Processing Components and Combinators

The set of fundamental components and combinators might include the ones below. As a typographical convention, different fonts are used for names of *components* and *combinators*. Many of these are common to a number of architectures, though the list here is far from complete and is only suggestive. Particular architectures might require other specialised concepts.

components: *action, activity, class, communication, constraint, identifier, information, interaction, interaction point, object, octet, parameter, provider, rendezvous, resource, state, state machine, template, term, type, user, value, variable.*

combinators: *aggregation, classification, configuration, conjunction, constraint composition, disjunction, hiding, instantiation, interleaving, interruption, iteration, mapping, negation, partial ordering, partition, relation, selection, sequence, set, state composition, string, supply composition, synchronisation.*

Space does not allow a full exposition of these fundamental concepts, but the following examples might be helpful:

components:

action: the establishment of one or more items of *information*.

activity: a *partial ordering* of *actions* by relative time of occurrence.

identifier: a unique label.

information: data with a predefined interpretation.

interaction: an *action* requiring the participation of two or more *objects*.

interaction point: a *partition* of *interactions* into groups representing distinct *communication*.

object: an *activity* and its associated *information*.

combinators:

hiding: making *interactions* invisible that are considered to be internal.

partial ordering: a partial, transitive *relation* over a *set*.

supply composition: an *interleaving* of *objects* considered as *users* in *synchronisation* with an *object* considered as *provider*.

The only unusual concept here is that of supply composition. This represents a common situation where a service is supplied to a number of users. The users are independent of each other and synchronise with the provider to obtain the service. The relationships among the fundamental concepts above are summarised in Figure 4. The arrows in this figure indicate how definitions rely on lower-level definitions; *identifier* is shown 'floating' as it does not depend on any other definition. Table 1 gives a general description of how the concepts might be represented in ESTELLE, LOTOS and SDL. In practice a full formalisation of each concept would be given, but to avoid technical details of three languages only a summary is given here.

2.4 OSI Architectural Semantics

As an example, a partial architectural semantics will be developed for OSI in relation to ESTELLE, LOTOS and SDL. OSI architectural concepts will be formulated in terms of the fundamental ones so that a simpler mapping can be defined. A more fully worked out exposition of OSI architectural semantics is given in [11]. Many useful hints are also given in the examples of [32]. A detailed treat-

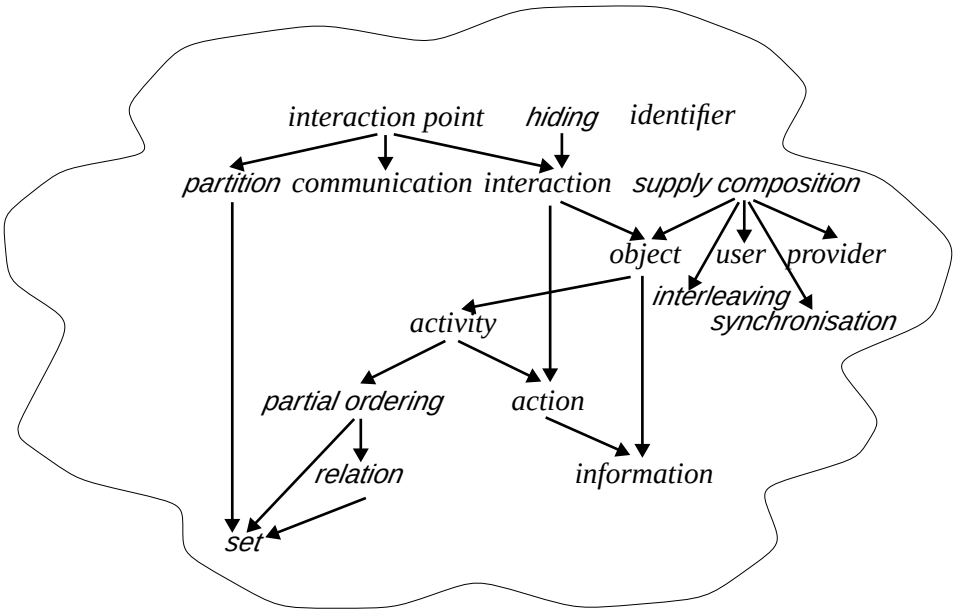

Figure 4 Relationships among Fundamental Concepts

Concept	**ESTELLE**	**LOTOS**	**SDL**
action	module transition	event (with parameters)	process transition
activity	module execution	behaviour	process execution
identifier	value of a designated type	value of a designated type	value of a designated type
information	PASCAL data type value	abstract data type value	abstract data type value
interaction	interaction	event synchronisation	signal
interaction point	interaction point	event gate	end of channel/signalroute
object	module instance	process instantiation	process instance
hiding	internal interaction point	hiding an event gate	internal channel/signalroute
partial ordering	module body	behaviour expression	process definition
supply composition	*interleaving* of *user objects* in *synchronisation* with a *provider object*		

Table 1 Representations of Some Fundamental Concepts

ment with respect to LOTOS is given in [29, 38]. The set of architectural components and combinators needed for OSI might include the following:

components:

address: an *identifier* for a *service access point*.

function: a self-contained *activity* of a *protocol entity.*

protocol entity: a *service user* capable of supporting *associations* with other *service users* of the same underlying *service provider.*

service access point: an *interaction point* between a *service user* and a *service provider.*

service data unit: *information* that may be conveyed in a *service primitive parameter* without interpretation by the *service provider.*

service primitive: an *interaction* between a *service user* and a *service provider.*

service provider: a *provider.*

service user: a *user.*

combinators:

association: a *partition* of *service user interactions* requiring *communication* via the *service provider* of *interactions* within the same *partition.*

blocking: a one-to-many *relation* between *service data units* of a *service user* and its supporting *protocol entity.*

multiplexing: a many-to-one *relation* between *associations* of a *service user* and its supporting *protocol entity.*

protocol: a *supply composition* of *protocol entities* considered as *users* and an underlying *service* considered as *provider.*

service: the *hiding* of *interactions* between the *protocol entities* and the underlying *service* of a *protocol*.

The concept of supply composition is used here to model the combination of protocol entities with an underlying service to produce a protocol. Hiding the internal service access points of this protocol yields a new service, so services are nested as would be expected in a layered architecture like OSI. Figure 5 illustrates the composition of a service; note that *service* and *protocol* refer to each other as the definitions are recursive. The relationships among the architectural concepts are summarised in Figure 6. For clarity, the use of fundamental concepts is shown separately in Figure 7; *protocol entity* is shown 'floating' as it does not rely on any fundamental concepts.

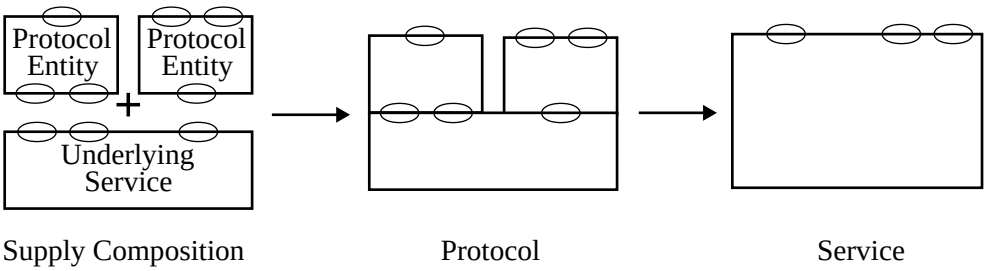

Figure 5 Composition of a Service

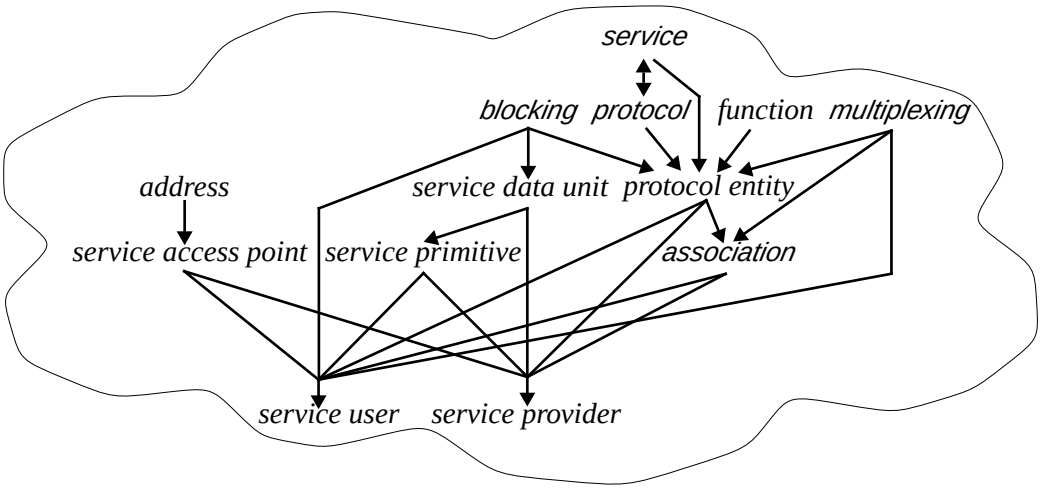

Figure 6 Relationships among Architectural Concepts

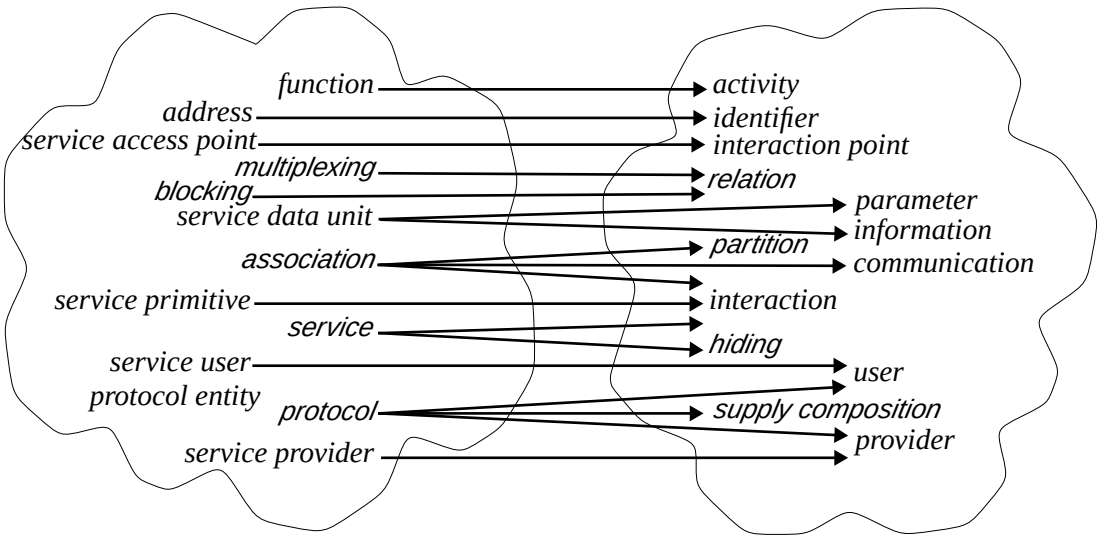

Figure 7 Relationships among Architectural and Fundamental Concepts

Table 2 suggests representations of the architectural concepts above in ESTELLE, LOTOS and SDL As earlier, the exact representation in each language is omitted for simplicity; in fact the representations shown are sufficiently general that their description applies to all three languages, so there is not one column per language. Note that the approach allows many of the denotations to be given in terms of the fundamental ones, thus simplifying the mapping. For concreteness, Section 3.3 shows how a selection of these concepts may be specified in LOTOS.

Concept	**ESTELLE, LOTOS, SDL**
address	*identifier* from a specially designated *type*
function	self-contained *activity* of a *protocol entity*
protocol entity	*user* supporting *associations*
service access point	*interaction point* between *user* and *provider*
service data unit	*information* represented as an *octet string*
service primitive	*interaction* between *user* and *provider*
service provider	*object* in a *provider* role
service user	*object* in a *user* role
association	*function* supporting a *partition* of *user interactions*
blocking	*function* supporting a one-to-many *relation* between *service data units*
multiplexing	*function* supporting a many-to-one *relation* between *associations*
protocol	*interleaving* of *protocol entities* in *synchronisation* with an *underlying service*
service	*hiding* of internal *protocol interaction points*

Table 2 Representations of Some Architectural Concepts

2.5 Other Architectural Semantics

An architectural semantics for ODP is in the process of being standardised [16, 27]. This work follows a broadly similar approach to the OSI architectural semantics, which in fact served as the trigger for it. The fundamental information processing concepts needed for ODP are a superset of those needed for OSI. The architectural semantics is actively being developed for LOTOS and Z, with other contributions using ESTELLE and SDL.

Design concepts for ODP have been investigated separately in [40]. An architectural semantics for distributed systems has also been investigated in [21]. The emphasis of this work was CIM (Computer Integrated Manufacturing), although making extensive use of OSI and ODP principles. Other issues such as real time, priority and probability were considered as well, using LOTOS as the basis.

The author's goal in [31] was to support the creation of communications services in a flexible, user-oriented and formal manner. The formulation of an architectural semantics proved helpful in defining the architecture of services as well as the mapping to LOTOS. For comparison with the OSI architectural semantics, some of the components and combinators needed for service creation are listed below without explanation. The basic components are taken to be service facilities; each may exist in five basic patterns, and each pattern has five possible ordering combinators. The combinators are ways of combing service facilities into more complex services. For fuller details see the original paper.

components: *provider-confirmed facility, provider-initiated facility, user/provider-confirmed facility, unconfirmed facility, user-confirmed facility.*

combinators: *alternate, colliding, consecutive, disables, enables, interleaves, interrupts, ordered, overtakes, reliable, simultaneous, single, unreliable.*

A rather different kind of architectural semantics was developed by the author for specification of digital logic [37]. Although digital logic may be loosely considered as an information processing system in the same sense as OSI, ODP or CIM, the range of architectural concepts needed is rather specialised. The basic concepts are rather simple, but the range of combinations is very large. Again for comparison with the OSI architectural semantics, the following bare list suggests some of the architectural concepts needed; the original paper should be consulted for a full explanation.

components: *binary signal, logic sink, logic source.*

combinators:

one-input: *delays, inverter, repeater.*

two/three/four-input: *and, nand, nor, or, xor.*

logic/arithmetic: *adders, coders, plexers.*

memories: *flip-flops, dividers, latches.*

3 Specification Templates

3.1 Motivation

Defining an architectural semantics establishes a sound relationship between an architecture and the language used to formalise it. However, as will have been evident from Section 2 the architectural semantics may be rather intricate. Certainly an architectural semantics in the form described there will not be of immediate practical use to the specifier. What is required is a codification of the architectural semantics to assist in producing compatible specifications of an architecture more directly. In the analogy of Section 1.3, what is needed is the equivalent of the electronic designer's handbook.

The concept of a **specification template** is therefore introduced. Several views can be taken of this. At a basic level, a specification template is simply a syntactic device. It is merely a fragment of specification text that can be conveniently recalled and inserted in a specification. To enhance the value of such templates they can be parameterised to increase their generality. Parameterisation might be at the level of the specification language, such as value parameters, variable names or process names. However, parameters could in principle take any syntactic form, such as pieces of behaviour, declarations or expressions. Indeed the power of templates comes as much from specifying combina-

tions as from specifying components. The emphasis on combinations reflects the fact that it is easier to fill in parts of a defined framework – the specification architecture.

From another viewpoint, specification templates embody component re-use. This has been a holy grail of software production for decades. Of course, the components used in a specification are likely to be very different from software components. Specification components will usually reflect abstract statements of requirements. They may simply be common data types needed to describe a system, such as addresses, protocol data units or service data units in a communications system. A specification component may also describe an abstract computation, such as routing, blocking or multiplexing in a communications system. More interestingly, specification components may also be logical statements defining constraints, assertions or invariants. The specification of a message-switching system, for example, might make use of generic requirements on reliable delivery of messages, correct ordering of messages or quality of service. These requirements could be formulated as specification fragments to be incorporated in a number of specifications.

Specification templates should ideally have an architectural meaning. The concepts of an architecture should correspond to templates that the specifier can call up. This would allow specification at a higher, architectural level. Consistency, productivity and comprehensibility would be improved by this. Templates should therefore be based on architectural semantics, making architectural specification relatively painless and immediate. However, it is not always useful to define a template for each architectural concept. In some cases the concept may be intermediate or auxiliary, serving to define a more specific concept. In other cases, the concept may be so general that a template would say little. In these circumstances it may be better to define templates only for more definite instances of the concept.

As possible templates consider protocol, supply composition and function. The architecture of a protocol as protocol entities composed with an underlying service can be given directly as a specification fragment; it appears in Section 3.3. On the other hand, the notion of supply composition is just a step towards defining a protocol and is not interesting as a template in its own right. Finally, a function is too general a concept to give as a template; only specific functions such as concatenation and multiplexing would be sensibly given as templates.

It may be convenient to focus verification on specification templates, emphasising their semantics rather than syntactic form. As [31] contends, success in engineering depends on combining known components in known ways to produce predictable results. In a specification context, templates should have known properties that have been verified in advance. The difficulties of verifying the overall specification might then be reduced by a hierarchical approach: known combinations of verified specification components leading to trusted components at a higher level.

Specification templates suggest the use of tools. A translator is required that will replace references to templates with the corresponding specification text. Such a translator might act as a property-oriented compiler – one that generates ‘code’ according to the required behavioural properties rather than according to a given algorithm. The use of templates can support a declarative style of specification, facilitating translation of architectural properties into the specification language [31].

3.2 Approach

The approach taken to specification templates is essentially textual. This makes a fairly safe presumption that the specification language has a textual form, but could be adapted to graphical forms of specification. Ideally, templates would be expressed directly in the specification language itself. To some extent this is possible, but unfortunately the flexibility needed in parameterisation and textual substitution is beyond convenient reach in most languages. It might be noted that programming languages are often used with a preprocessor in order to achieve this kind of flexibility. The preferred approach is therefore to express templates in a form that is preprocessed into the specification language.

A macro processor rather than a purpose-built translator is an obvious choice for dealing with preprocessing of specification templates. Macro files can act as specification libraries, and can be used

almost invisibly by the specifier. For example, LITE (LOTOSPHERE Integrated Tool Environment [39]) allows LOTOS specifications to be preprocessed automatically. This might include running a macro processor to instantiate specification templates. A macro-based approach does not in itself confer many benefits; the added value comes from linking templates to architectural concepts, architectural semantics and verification procedures.

The particular choice of macro language is not important provided it allows adequate parameterisation. As will be described, the author has created several libraries of specification templates using the *m4* macro processor [20: Chapter 8]. The main reason for choosing *m4* is its widespread availability; it has been distributed with UNIX for many years and can also be obtained for other environments. In brief, *m4* allows (parameterised) macros, conditional macro expansion and string operations on input. These basic facilities are not immediately useful to build a typical specification template library. However, with care and ingenuity, higher-level macro facilities can be defined [33]. These include text variables, hierarchies of macros, loops (using recursion), higher-order macros (taking macros as parameters) and 'curried' macros (partially applied to some of their parameters). Creating a template library is rather easier with these facilities, though it must be admitted that the work is rather intricate.

A template library must exist for each architecture-language pair. Fortunately, a basis in architectural semantics reduces the work involved in building such libraries. Indeed, definition of an architectural semantics and a template library can usefully proceed together. By focusing attention on architectural issues, the specifier can think in problem-oriented rather than language-oriented terms. If the template library were not defined formally, there might be loss of precision. However if the library is just a textual veneer on top of the language, the denotation of expressions using the library should be immediate and obvious.

A possible problem might be the indirect link between the specification using templates and the specification using the language proper. As will be seen, the use of templates can result in 'specifications' that are a few percent of the true specification's size after macro expansion. This is a good indication of the productivity and effectiveness of specification templates. But the specifier must ultimately work with the generated specification, say to refine, verify or implement it. Template expansion is thus unlike conventional compilation, where the code resulting from translation is of little interest to the programmer. The structure of the template library and its expansion through macro processing must therefore be clear to the specifier. This is where the emphasis on architectural issues again plays an important role.

3.3 OSI Specification Templates for LOTOS

As an example, the partial architectural semantics considered for OSI in Section 2.4 will be revisited. For brevity a selection of specification templates for only LOTOS will be discussed, though a similar treatment could be given for ESTELLE or SDL.[5] Familiarity with LOTOS will be assumed. The facilities of *m4* used in the templates will be described as they appear.

A service provider sees a service data unit as an unstructured sequence of octets. The template defining it is therefore a straightforward renaming of the library type for an octet string:

define(data_type,'
  type *DATA* ***is*** *OctetString* ***renamedby***
    sortnames *Data* ***for*** *OctetString*
  endtype *(* DATA *)*
')

Since there are two languages used here they are distinguished by typeface, with *m4* given in roman type and LOTOS in italics. Keywords in both languages are emboldened. In *m4*, **define** introduces a macro name (data_type) and its definition ('...'). Macro definitions are usually given in single quotes

[5]This would use the textual form of SDL/PR rather than SDL/GR.

to prevent their interpretation prior to macro expansion. The above template for service data units can be included in a LOTOS specification simply by giving its name. The template defines constant text and is unparameterised. In fact, templates realising LOTOS data types are mainly unparameterised.

For addresses, the architectural semantics simply requires a set of distinct values. Although this could be achieved by renaming the natural number type, this would introduce additional operations (like +) that would not be meaningful for addresses. Addresses simply have the form of a base address or an address 'following' another one:

```
define(addr_type,'
  type ADDR is Boolean
    sorts Addr
    opns
      BaseAddr :                    → Addr
      AnotherAddr :     Addr        → Addr
       _eq_,_ne_ :      Addr, Addr  → Bool
    eqns
      forall addr1, addr2: Addr
        ofsort Bool
          BaseAddr eq BaseAddr                          = true;
          AnotherAddr (addr1) eq BaseAddr               = false;
          BaseAddr eq AnotherAddr (addr2)               = false;
          AnotherAddr (addr1) eq AnotherAddr (addr2)    = addr1 eq addr2;
          addr1 ne addr2                                = not (addr1 eq addr2);
  endtype (* ADDR *)
')
```

Other templates would generate addresses from this one for the upper and lower boundaries of a protocol entity. Another template would generate sets of addresses.

A service data unit may be blocked with others into a protocol data unit; the inverse operation at the receiver is deblocking. The OSI architecture states that blocking and deblocking are inverse operations, but does not prescribe the manner in which they are carried out; this is left to individual protocol standards. The LOTOS specification of (de)blocking defines data type operations on service data units. Deblocking in fact requires two operations: one to extract a service data unit embedded in the whole protocol data unit, and one to extract the remainder of the protocol data unit. In the following definition, <> is an empty string.

```
define(block_type,'
  type BLOCK is DATA
    opns
      deblock_pdu :     Data        → Data
      deblock_sdu :     Data        → Data
      block_pdu :       Data, Data  → Data
    eqns
      forall pdu: Data, sdu, sdu1, sdu2: Data
        ofsort Data
          deblock_pdu (<>)                                  = <>;
          deblock_pdu (block_pdu (<>, sdu))                 = <>;
          deblock_pdu (block_pdu (block_pdu (pdu, sdu1), sdu2)) =
            block_pdu (deblock_pdu (block_pdu (pdu, sdu1)), sdu2);
          deblock_sdu (<>)                                  = <>;
          deblock_sdu (block_pdu (<>, sdu))                 = sdu;
          deblock_sdu (block_pdu (block_pdu (pdu, sdu1), sdu2)) =
            deblock_sdu (block_pdu (pdu, sdu1));
```

```
  endtype (* BLOCK *)
')
```

Service primitives present more of a challenge. The occurrence of a service primitive is simply a LOTOS event with the following parameters: the event gate carrying communication with the service, the identification of the user, and the service primitive information. Such an event can be defined with the template:

```
define(serv_prim,‘
  $1 ! $2 ! $3 ;
’)
```

Template parameters are numbered $1, $2, … Their use here corresponds to a call of the form serv_prim(gate,identifier,information) which generates the LOTOS action prefix:

gate ! identifier ! information ;

For services that support multiple associations the identifier has two components, one for the service access point and one for the association endpoint within the service access point.

The most involved aspect of service primitives is the structure of information they convey. There cannot be a fixed template for this because of the variety of service primitives and their parameters. Specifying service primitives directly in LOTOS is tedious and error-prone. A parameterised template with the names of the service primitives and their parameter types allows automatic specification of primitives.

Suppose, for example, that a connection-oriented service has user-confirmed connection, unconfirmed normal data transfer, unconfirmed expedited data transfer, and unconfirmed user disconnection. The following template instance defines service primitives and appropriate parameters for this service:

```
prim_type(
  Conn_Request(Addr1,Addr2)  Conn_Indication(Addr1,Addr2)
  Conn_Response(Addr)        Conn_Confirm(Addr)
  Data_Request(Data)         Data_Indication(Data)
  Exped_Request(Data)        Exped_Indication(Data)
  Disconn_Request(Reason)    Disconn_Indication(Reason)
)
```

The prim_type template is a relatively complex set of *m4* definitions that take the service primitive declarations and turn them into a composite LOTOS type. In fact there are a number of subsidiary templates ‘behind the scenes’ to specify the primitive data type from this information. The parameters of service primitives are used to derive the signatures of the corresponding LOTOS operations and to derive their equations. The *m4* definitions needed are too detailed to be given here, so only the translation of the prim_type call above will be given – and even that in outline only:

```
type PRIM is ADDR,DATA,REASON
  sorts Prim
  opns
    Conn_Request:        Addr, Addr    → Prim
    Conn_Indication:     Addr, Addr    → Prim
    Conn_Response:       Addr          → Prim
    Conn_Confirm:        Addr          → Prim
    Data_Request:        Data          → Prim
    Data_Indication:     Data          → Prim
    Exped_Request:       Data          → Prim
    Exped_Indication:    Data          → Prim
    Disconn_Request:     Reason        → Prim
    Disconn_Indication:  Reason        → Prim
```

```
    Is_Conn_Request, Is_Conn_Indication, Is_Conn_Response, Is_Conn_Confirm,
      Is_Data_Request, Is_Data_Indication, Is_Exped_Request, Is_Exped_Indication,
        Is_Disconn_Request, Is_Disconn_Indication:
                                  Prim          → Bool
    Is_Request, Is_Indication:    Prim          → Bool
    _eq_, _ne_:                   Prim, Prim    → Bool
endtype (* PRIM *)
```

The type *PRIM* is of course part of the prim_type template, but its definition has to be derived from the parameters and so cannot be given literally.

Architectural combinators usually appear in process definitions. According to the architectural semantics, a protocol consists of a set of interleaved protocol entities synchronised with an underlying service. The template for this might include the following process definition:

```
process Protocol [$1, $2] ($1_Addrs: $1_Addr_Set) : noexit :=
  choice $2_Addrs: $2_Addr_Set []
    prot_ents($1, $2, $1_Addrs, $2_Addrs)
  |[$2]|
    under_serv($2, $2_Addrs)
endproc (* Protocol *)
```

Parameters $1 and $2 give the names for the upper and lower protocol gates; these also qualify the address names. Note the combination of calls on subsidiary templates to define the protocol entities (prot_ents) and the underlying service (under_serv). The whole template might be expanded to the following definition for the protocol process:

```
process Protocol [Upper, Lower] (Upper_Addrs: Upper_Addr_Set) : noexit :=
  choice Lower_Addrs: Lower_Addr_Set []
    Protocol_Entities [Upper, Lower] (Upper_Addrs,Lower_Addrs)
  |[Lower]|
    Underlying_Service [Lower] (Lower_Addrs)
endproc (* Protocol *)
```

The template library takes care of a small problem with templates like this. In the protocol template, process *instantiations* must be generated by the subsidiary templates prot_ents and under_serv. However, process *definitions* are also needed somewhere so the templates generate these separately.

The final example of a specification template is for a service. This is obtained by hiding the lower service access points (and hence gate) of a protocol:

```
process Service [$1] ($1_Addrs : $1_Addr_Set) : noexit :=
  hide Lower in
    prot($1,Lower)
endproc (* Service *)
```

The foregoing templates are a small part of an OSI template library for LOTOS developed by the author. The template library has about 70 *m4* macros occupying 530 non-comment lines, and is available on-line as [35]. To appreciate how such a library might be useful, consider the connection-oriented service example used to illustrate service primitives above. The description using the template library is as follows:

```
co_serv_spec(
  CO,
  Conn_Request(Addr1,Addr2)  Conn_Indication(Addr1,Addr2)
  Conn_Response(Addr)        Conn_Confirm(Addr)
  Data_Request(Data)         Data_Indication(Data)
  Exped_Request(Data)        Exped_Indication(Data)
```

```
    Disconn_Request(Reason)      Disconn_Indication(Reason)
)
```

This is virtually identical to the call of prim_type except that the communication interface CO (a LOTOS gate) is given as a parameter. When translated to LOTOS the resulting specification is about 490 lines. A 1:50 expansion is fairly typical of the productivity gains that are possible using specification templates.

3.4 Other Specification Templates

Template libraries have been developed for some of the other architectural semantics discussed in Section 2.5.

For the ODP architectural semantics, [41] has advocated the use of LOTOS templates. The intent of this work is similar to that described here, although expansion of templates through a macro library was not the main point. This, however, would seem to be a straightforward extension.

The components and combinators for service creation have been embedded in *m4* templates that support a special-purpose specification language called SAGE (Service Attribute Generator [31]). The pattern and ordering property of a service facility are translated into a LOTOS specification fragment. Combinations of facilities are realised using LOTOS temporal operators in predefined arrangements. As a simple example of these templates, consider an acknowledged connection-less service. This supports one-way transfer of datagrams carrying source/destination addresses and user data. Transfer of datagrams is confirmed by the service provider and is reliable. The following template instances declare these properties:

forall(**facility**(12,provider_confirmed,reliable,Datagram(Addr,Addr,Data)))

The **facility** template defines a service facility; the notation 12 gives the direction of transfer (from user 1 to user 2). The **forall** declaration permits all instances of the datagram facility. The service creation library holds about 35 macros in about 650 non-comment lines of *m4* and is available on-line as [36]. Full details of the approach and other examples are given in the original paper.

The architectural semantics defined for digital logic has been incorporated into a template library called DILL (Digital Logic in LOTOS [37]). This allows an electronics designer to formulate a circuit using conventional components and to have a LOTOS specification generated automatically. The specification can be generated and simulated with only limited knowledge of LOTOS. A further development of this work allows graphical animation of the circuit design, so the electronics designer needs only a superficial understanding of LOTOS [22]. The digital logic library holds about 65 macros in about 600 non-comment lines of *m4* and is available on-line as [34].

A small example of a circuit might be a two-input ‘not and’ function that realises the logical function $\neg IP_1 \wedge IP_2$. This is not commonly available as a ready-made component, but ‘not’ and ‘and’ functions are. The circuit may be constructed from an inverter and a two-input ‘and’ gate. The template instance for this is given in Figure 8 along with the conventional circuit diagram.

```
circuit(‘NotAnd2 [Ip1, Ip2, Op]’,‘
  hide NotIp1 in
    Inverter [Ip1, NotIp1]
  |[NotIp1]|
    And2 [NotIp1, Ip2, Op]
  where
    Inverter_Decl
    And2_Decl
’)
```

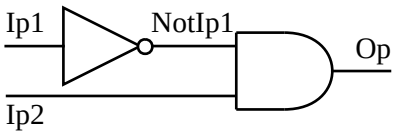

Figure 8 Composition of a Circuit

Logic gates are ‘wired up’ using the LOTOS synchronisation operator. The component library described in the original paper supports declarations such as Inverter_Decl and And2_Decl.

4 Conclusion

It has been argued that the use of multiple specification languages with multiple architectures may create problems of inconsistency. The development of appropriate specification styles for all these languages and architectures could be time-consuming. The architectures themselves may be loosely defined, hindering formalisation. An approach to defining architectural semantics via fundamental information processing concepts has therefore been advocated, offering a solution to these problems. It has been shown how architectural semantics can be embodied in a library of specification templates as a practical tool for the specifier. OSI has been used as the main example, with other illustrations taken from ODP, CIM, service creation and digital logic. Although all of these architectural semantics could be elaborated further, investigations have been conducted in sufficient breadth and depth to give confidence in the value of the approach.

Acknowledgements

Most of the work on OSI architectural semantics took place under the chairmanship of the author in FDT sub-group A. Contributors to the this work included David Freestone (British Telecom), Dieter Hogrefe (University of Hamburg[6]), Giuseppe Scollo (University of Twente), Chris Vissers (University of Twente) and Fritz Vogt (University of Hamburg). The architectural semantics and template library for digital logic were jointly investigated with Richard Sinnott (University of Stirling), who was supported at the time by a studentship from the UK Science and Engineering Research Council. Helpful comments on a draft of the paper were received from Richard Sinnott (University of Stirling). Penetrating observations by the anonymous reviewers were particularly valuable in improving the paper.

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] Michael J. C. Gordon. HOL: A proof generating system for Higher-Order Logic. Technical Report 103, University of Cambridge, UK, 1987.

[4] 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.

[5] Reinhard Gotzhein. *Open Distributed Systems: On Concepts, Methods, and Design from a Logical Point of View.* PhD thesis, University of Hamburg, 1992.

[6] C. Anthony R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1985.

[7] IEEE. *VLSI Hardware Design Language*. IEEE 1076. Institution of Electrical and Electronic Engineers, New York, USA, 1992.

[6] Now University of Lübeck.

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

[9] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – ESTELLE – A Formal Description Technique based on an Extended State Transition Model*. ISO/IEC 9074. International Organization for Standardization, Geneva, Switzerland, 1989.

[10] 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.

[11] 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.

[12] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*. ISO/IEC 8825. International Organization for Standardization, Geneva, Switzerland, 1990.

[13] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*. ISO/IEC 8824. International Organization for Standardization, Geneva, Switzerland, 1990.

[14] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*. ISO/IEC 9646-3. International Organization for Standardization, Geneva, Switzerland, 1991.

[15] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conventions for the Definition of OSI Services*. ISO/IEC TR 10731. International Organization for Standardization, Geneva, Switzerland, 1992.

[16] 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.

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

[18] ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 1992.

[19] Clifford B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1990.

[20] Brian W. Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley, Reading, Massachusetts, USA, 1976.

[21] Ashley McClenaghan. Distributed systems: Architecture-driven specification using extended LOTOS. Technical Report CSM-120, Department of Computing Science, University of Stirling, UK, December 1993.

[22] Ashley McClenaghan. XDILL: An X-based simulator tool for DILL. Technical Report CSM-119, Department of Computing Science, University of Stirling, UK, April 1994.

[23] George J. Milne. The formal description and verification of hardware timing. *IEEE Transactions on Computers*, 40(7):811–826, July 1991.

[24] A. J. Robin G. Milner. *Communication and Concurrency*. Addison-Wesley, Reading,

Massachusetts, USA, 1989.

[25] RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1992.

[26] 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.

[27] Richard O. Sinnott. The development of an architectural semantics for ODP. Technical Report CSM-121, Department of Computing Science, University of Stirling, UK, March 1994.

[28] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1992.

[29] Kenneth J. Turner. LOTOS – A practical Formal Description Technique for OSI. In *International Open Systems 87*, volume 1, pages 265–279. Online Publications, London, March 1987.

[30] 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.

[31] 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.

[32] Kenneth J. Turner, editor. *Using Formal Description Techniques — An Introduction to ESTELLE, LOTOS and SDL*. Wiley, New York, January 1993.

[33] Kenneth J. Turner. Exploiting the *m4* macro language. Technical Report CSM-126, Department of Computing Science, University of Stirling, UK, September 1994.

[34] Kenneth J. Turner. DILL (Digital Logic in LOTOS) translator. http://www.cs.stir.ac.uk/~kjt/software/lotos.html, April 1996.

[35] Kenneth J. Turner. BASIL (Basic Architectural Semantics in LOTOS) translator. http://www.cs.stir.ac.uk/~kjt/software/lotos.html, April 1996.

[36] Kenneth J. Turner. SAGE (Service Attribute Generator) translator. http://www.cs.stir.ac.uk/~kjt/software/lotos.html, April 1996.

[37] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.

[38] Kenneth J. Turner and Marten van Sinderen. LOTOS specification style for OSI. In Jeroen van de Lagemaat, Tommaso Bolognesi, and Chris A. Vissers, editors, *The LOTOSPHERE Project*, pages 137–159. Kluwer Academic Publishers, London, UK, 1995.

[39] Peter H. J. van Eijk. The LOTOSPHERE integrated tool environment LITE. In Kenneth R. Parker and Gordon A. Rose, editors, *Proc. Formal Description Techniques IV*, pages 471–474. North-Holland, Amsterdam, Netherlands, November 1991.

[40] Marten van Sinderen, Luís Ferreira Pires, and Chris A. Vissers. Design concepts for Open Distributed Systems. In Jan de Meer, B. Mahr, and Otto Spaniol, editors, *Proc. Int. Conf. on Open Distributed Processing*, pages 369–374, Berlin, Germany, September 1993. Gesellschaft für Mathematik und Datenverarbeitung.

[41] Andreas Vogel. On ODP's architectural semantics using LOTOS. In Jan de Meer, B. Mahr, and Otto Spaniol, editors, *Proc. Int. Conf. on Open Distributed Processing*, pages 340–345, Berlin, Germany, September 1993. Gesellschaft für Mathematik und Datenverarbeitung.