Kenneth J. Turner. An engineering approach to formal methods. In Andre 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.

An Engineering Approach to Formal Methods

K. J. Turner[a]

[a] Department of Computing Science, University of Stirling, Stirling FK9 4LA, Scotland

The distinctive features of engineering are discussed, and used to identify how an engineering approach to formal methods might be developed. The key concept in engineering is suggested to be known components that are combined in known ways. This component-based style is illustrated for two application areas at two levels: in high-level specification of communications services, and in low-level specification of digital logic. The underlying formal language is LOTOS (*Language Of Temporal Ordering Specification*).

Keyword Codes: B.2.1; B.6.1; C.2.1; D.2.1; F.4.3

Keywords: Arithmetic and Logic Structures, Design Styles; Logic Design, Design Styles; Computer Communication Networks, Network Architecture and Design; Software Engineering, Requirements/Specifications; Mathematical Logic and Formal Languages, Formal Languages; Communications Services; LOTOS (Language Of Temporal Ordering Specification)

1 Introduction

The topic of formal methods covers the development and application of mathematically-based approaches in computing. But is it a science, an engineering discipline, or both?

There is growing interest in formal methods because they offer rigorous support of computer system development. Formal methods are particularly desirable in safety-critical applications such as process control, aviation, medical systems, railway signalling and many others. Other applications may not threaten life if they fail, but most may be described as quality-critical. It is difficult to find an application that would not benefit from the rigour brought by formal methods. However, the main reason that formal methods are limited in their use is that on a cost-benefit analysis they are often not justified. The only way to make them more widely applicable is to reduce the cost of their use.

Engineers make successful use of science to achieve practical results. There is reason to believe that a combination of engineering principles and formal methods could lead to rigorous and cost-effective computer system design. Section 2 investigates what is distinctive about engineering and what its lessons are for formal methods. A key aspect of success in engineering is suggested to be a component-based style in which known components are combined in known ways to yield predictable results. Section 3 illustrates the approach by showing how high-level specifications of communications services can be produced. Section 4 illustrates the approach in a different application area by showing how to produce low-level specifications of digital logic. In both cases, the underlying formal language is LOTOS (*Language Of Temporal Ordering Specification*, [2]).

2 An Engineering Approach to Formal Methods

This section addresses a number of aspects of general engineering practice, and suggests some implications for engineering with formal methods.

2.1 The Place of Formal Methods

Mathematics is widely used in all aspects of engineering. However, it is taught in an applied way and is backed up by well-defined methods. The mathematics is packaged in a form directly usable by an engineer. Often the notation and the results rather than the underlying theory are the important parts of the mathematics. Formal methods in computing should aspire to the same level of utility and acceptability. Fortunately, there is good evidence that this can be achieved. One good example, which is often overlooked, is the theory of artificial languages. Every compiler writer uses this in parsing and processing languages, and every programmer is accustomed to at least the grammar of a language.

So what is it that distinguishes engineering from science? In general, science is concerned with explanation. A typical dictionary definition of science is 'knowledge covering general truths or the operation of general laws' [5]. Science thus deals with fundamental ideas and theories. Science is often analytic, seeking to understand phenomena in terms of underlying explanations. By way of contrast, a typical dictionary definition of engineering is 'the application of science and mathematics . . . made useful to people' [5]. Engineering is thus concerned with application or production. Engineering puts scientific results to practical use. Engineering is often synthetic, building new solutions from existing ones.

Trying to polarise science and engineering is artificial. There are many scientists who carry out engineering activities, and many engineers who carry out scientific investigations. There is a full range from pure science without any applications to pure engineering without any scientific basis. But it is useful to compare the opposite ends of the range in order to see how they differ.

Science and engineering are well-established disciplines, in some cases going back millenia. Computing goes back only 30 to 50 years, so of course the body of scientific and engineering knowledge in this area is still growing enormously. Computer science may be seen as the scientific branch of computing. Computer science has been able to draw considerably on work in the physical and numerical sciences (e.g. physics, electronics and mathematics). Theoretical computer science focusses especially on the mathematical underpinning of computation. On the engineering side of computing, there is a split into hardware engineering and software engineering. Hardware engineering has been developed directly on top of electronic engineering. Formal methods in hardware engineering are well-advanced (e.g. hardware description languages, design automation systems and simulation systems). However, software engineering has had very little to build on; the concept of software (though not algorithm) hardly existed before the 1950s.

So, where do formal methods fit in? Part of the 'image' problem they have is that they are seen largely as a scientific pursuit. Formal methods are seen as being rather mathematical and intellectually hard to use. They have a reputation of being abstruse and impractical. They have limited use in industry, though the use that is reported is generally favourable.

Figure 1 relates some of the areas discussed so far. Formal methods includes theoretical computer science, formal software engineering and formal hardware engineering. Of these three, formal software engineering requires the most attention. Focussing attention on formal software engineering will hopefully identify weaknesses and areas where work is needed.

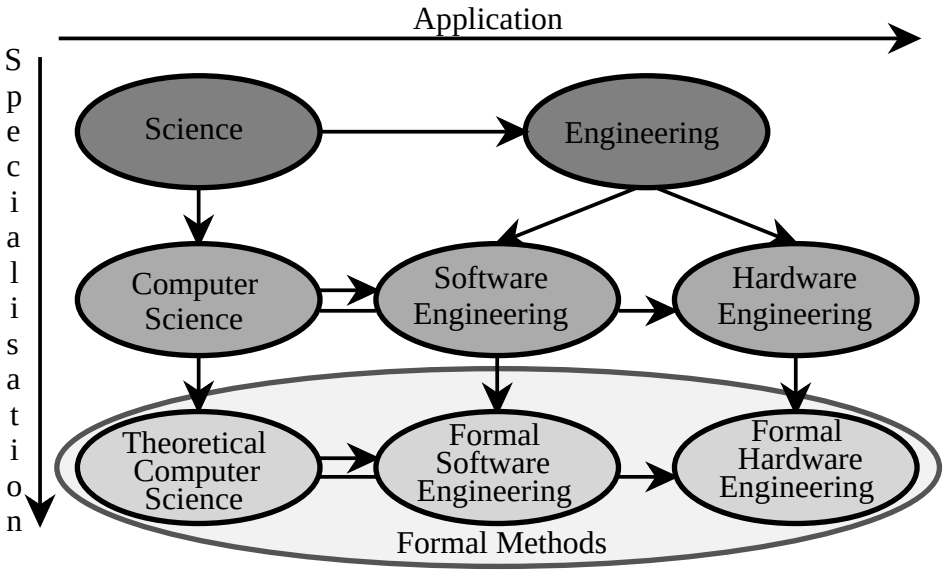

Figure 1. The Place of Formal Methods

Trying to draw a fixed boundary between hardware engineering and software engineering is, of course, as hopeless as trying to draw a fixed boundary between science and engineering. Hardware may have integral software support (e.g. the microprogram of a microprogrammable processor), and software may have integral hardware support (e.g. bit block-level transfer in a windowing system). Nonetheless, hardware and software differ fundamentally in how they are designed, manufactured, enhanced and repaired. By the same token, formal methods in hardware engineering and software engineering have evolved along different lines. As it happens, formal methods in hardware engineering are much better developed, so the following discussion mainly addresses formal software engineering.

2.2 An Engineering Approach

2.2.1 Engineering Philosophy

Engineers aim to use scientifically-based methods and tools. Certainly there are times when a particular engineering problem does not have a scientific underpinning. In the main, however, engineering is concerned with the application of science. Engineers also aim to produce practical results that solve real problems, generally industrial problems in manufacturing or construction. Although new problems may require new solutions, much of engineering is the re-use of trusted components, methods and tools. An engineer can call on a large body of experience that indicates the costs and benefits of each possible solution.

Formal software engineering should similarly aim to apply methods and tools based on the results of theoretical computer science. The goals should therefore be directly practical, aiming to build better software in a predictable manner. Like all engineers, formal software engineers should be actively involved in solving industrial problems. However, formal software engineers currently lack the historical experience and case studies available to other engineers. This is largely due to the immaturity of the subject, and so will improve in time anyway. Nonetheless, case studies and evaluations of methods and tools should be actively pursued. External influences

such as standards and Government-promoted initiatives are important incentives to employ more advanced techniques.

Engineers are expected to follow approved practices, using appropriate components and tools. Results are expected to be predictable and to meet constraints such as cost, time, quality, reliability and safety. Engineers have to behave professionally, and often require to belong to a professional society and have chartered status before they can practise.

In principle, the same expectations could be held of formal software engineering. Unfortunately, the state of the art does not yet allow development to meet the kind of criteria that would be applied in other engineering. Software estimation techniques allow some prediction of cost and time, but they are still somewhat inexact. With physical systems, it is usual to predict failure rates and component lifetimes. Software does not, of course, wear out or break down in the same way as physical components. Work on software reliability will in future allow predictions of problems with software. Much more effort is needed, however, on predicting reliability and on designing to meet safety standards. Advances in areas like these may cause software engineers to be held individually accountable for their work. Perhaps software engineers of the future should be chartered like other engineers, and risk losing their charter if they are found to be professionally negligent.

2.2.2 Engineering Processes

An engineering process will generally follow an established approach to design and manufacturing. From many similar projects, an appropriate model of the development process will be selected. Standard project management methods will be used, backed up by quality control during design and manufacturing. Suitable tools will generally be available already.

The software engineering process has been described in many ways. However, few of these deal specifically with the use of formal methods in software engineering. For example, the shape of a development process using formal methods is rather different: much of the effort is up front on specification and verification, while rather less is devoted to testing and maintenance. The problems of managing formal software engineering derive in part from the lack of suitable metrics. Managers of software projects are accustomed to measuring numbers like lines of code produced, faults found during review, or modules whose testing is complete. Metrics for formal methods are still to be defined: number of specification lines, assertions, or theorems proved? The seamless use of formal methods throughout the development process also needs much more work. Unlike tools used in other forms of engineering, tools for formal software engineering are largely research prototypes and lack industrial applicability.

Models in engineering have traditionally been scale models or mock-ups. These are used to predict the behaviour of the real artifact. During evaluation, the model is adapted until the desired behaviour is obtained. More recently, engineers have turned to computer simulations as being more cost-effective than physical models.

Formal software engineers also build models of systems, but mathematical abstractions rather than scale models. Again, the object is to predict the behaviour of the real system. However, formal models are largely concerned with functionality, whereas engineering models generally are largely concerned with performance and reliability. Perhaps this is because engineering functions are often straightforward, whereas computing functions are rarely so. Predictions from mathematical models in computing are therefore largely related to correct behaviour. Formal methods that combine functional and non-functional aspects have already appeared, but

will need much more development.

2.2.3 Engineering Components and their Combination

Monolithic or amorphous systems are rare, except in nature. Design is almost invariably decompositional (top-down) or compositional (bottom-up). Engineering exploits this by aiming to use common components in different designs. For example, an electronics engineer uses off-the-shelf discrete or integrated components. Mass production of specialised components enables engineers to design new products quickly and effectively. Components are designed and manufactured to defined interfaces and standards, enabling them to be assembled with confidence into more elaborate structures. Extensive use of standardised components constrains designs, but at the same time limits variations that might not be cost-effective. For example, an electronics engineer designing computer memory will use standard chips that dictate memory size and word length, rather than trying to design an arbitrary memory structure.

Another important aspect of engineering is that ready-made solutions (or designs) are generally available. These combine known components in known ways to achieve predictable results. For example, an electronics engineer who wishes to build a parallel adder is likely to use standard components configured according to the circuit diagram in a standard reference book.

Component re-use has been a major theme in software engineering for many years. Object-oriented methods and languages seem to be the first practical step towards achieving this goal. However, in formal software engineering there has been little identification of useful specification components and specification structures using these. This is a great pity since a major promise of formal methods is verification of the system being specified. Verification is very hard for any but trivial systems, so verification of large or complex systems is usually infeasible in practice. A component-based style allows components to be verified individually. Larger combinations ('designs') of trusted components can then be verified more easily.

2.3 Key Aspects of Engineering

The preceding discussions has identified a number of suggestions for developing an engineering approach to formal methods. Of these, the key aspect seems to be using known components, in known combinations, supported by effective tools.

Ideally it should be possible to develop and prove components and combinations individually. In practice, this would allow components to be designed by third parties or bought in. It would also allow general-purpose designs to be evolved, and documented in standard reference works. It must also be possible for components to be combined without adversely affecting their individual properties. The ideal component is general enough to allow re-use or simple adaptation for new applications. At the same time, the component must not be so general as to make it expensive or to make customisation difficult.

In formal software engineering, there are two principal levels at which a component-based style of specification is particularly worthwhile: at a high level (close to requirements) and at a low level (close to implementation). For each application area and level of specification, a library of components and combinations should be developed. This can also help to bridge the gap between the customer or end-user, the specifier and the implementer. Specifications are usually couched in a specialised language, reflecting the features and concerns of that language. With a component-based style, there is an opportunity to impose structure on a specification that is meaningful to end-users.

A component-based style also allows the specifier to take a higher-level, architectural view

of the specification. This makes it easier to produce new specifications of similar problems, ensures greater consistency in style among different specifications in the same application area, and allows different specifications to be composed more easily. In the field of communications systems, the term 'architectural semantics' is also used for this approach [7]. This permits an architectural view of how a language should be used, restricting its usage but also making its use more evident.

Are component-based specifications designs? All specifications must exhibit structure unless they are monolithic, so large specifications should follow stylistic principles to ensure a good structure. A component-based style is simply one way of structuring specifications, and so does not necessarily lead to designs. As will be seen in section 3, components and combinations can be constraints or assertions, leading to high-level specifications. As will also be seen in section 4, components and combinations can also be detailed and concrete, leading to low-level specifications. The choice of components and combinations depends on the application area and the purpose of specifications. A range of abstraction levels should be used during the design trajectory, appropriate to each stage in development.

The remainder of this paper illustrates a component-based style of specification at a high level and at a low level. The first application deals with communications services, and might be considered as engineering with constraints. The second application deals with digital logic, and might be considered as engineering with physical components. In both applications, the important issue is the use of known components and combinations. These are backed up by a formal representation. LOTOS has been used in this paper, but in principle any formal notation could be used.

3 Engineering Communications Services

3.1 Service Engineering

The concept of service engineering is used in telecommunications, where there is increasing demand for rapid introduction of new services. The term service is used with at least two different meanings: as a set of functions performed on behalf of customers, and as the abstraction of the functions of a layer. The first meaning is the one used in ODP (*Open Distributed Processing*), the second is the one used in OSI (*Open Systems Interconnection*). Service engineering has previously been used of services in the first sense. However, this paper concentrates on the second meaning of service, largely because it is a much more structured and well understood problem domain. The specification of such services might also be legitimately termed service engineering. The goals are to reflect user requirements closely, to use well-known patterns of behaviour, to allow flexible definition and modification of services, and to formalise and verify the resulting services. A restriction imposed in this paper is that services are provided between pairs of users. However, the approach taken could be generalised to deal with multi-way (multi-peer) services.

OSI views a service as a collection of service facilities. The exact nature of these is left open, but the intention seems to be that service facilities should be self-contained features. For example, a simple connection-oriented service might be said to have facilities for connection establishment, data transfer and connection release.

Much experience has been gained in writing specifications of communications services in LOTOS. Guidance is available in documents such as [3, 8, 10]. The usual advice is to adopt

a constraint-oriented style, decomposing the service behaviour into endpoint constraints and end-to-end constraints. Unfortunately this decomposition makes the division into facilities a secondary concern. The behaviour of facilities is therefore scattered across the specification, making it difficult to add, change or remove facilities. A more convenient division would make decomposition into facilities the primary split, with consideration of other constraints secondary. This is the rationale behind the component-based style that is explained below.

3.2 Service Components

3.2.1 Service Primitives and Facilities

Service facilities are the components of services. Service facilities may be combined into larger facilities, so a service is effectively just the top-level facility. Service facilities correspond to patterns of interactions between a pair of users. The interactions correspond to the occurrence of service primitives. Service primitive occurrences are abstractions of interactions between a service user and a service provider. Service primitives are named according to the layer involved, the facility being invoked, and the role of the service primitive in the facility. A typical service primitive might thus be named *N-Connect request*, being a request by a network layer user to establish a connection. In the following, the layer prefix will be omitted as being implicit. Four roles are identified for service primitives in a facility:

request: this initiates some facility (e.g. to request a connection to another user)

indication: this notifies the corresponding user that the facility has been invoked (e.g. to notify a user that a connection has been requested)

response: this gives the acknowledgement from the responding user (e.g. to indicate acceptance of the connection)

confirm: this gives the acknowledgement to the initiating user (e.g. to indicate that the connection has been accepted).

A particular facility may require only some of these roles. Also, a facility might be subdivided into two: a request and indication, followed by a request and indication in acknowledgement. For example, a data request and indication might trigger an optional acknowledgement request and indication rather than a data response and confirm. In such a case, however, there are really two facilities: an unconfirmed data transfer and a confirmed one, selected according to some option in the data request.

A service primitive with name like *Connect request* belongs with others of the same facility in a group with name *Connect*. The group name is a label for the request, indication, response and confirm primitives collectively. If a request and indication rather than response and confirm are used in the acknowledgement, different group names are used. Thus a confirmed data transfer facility might be subdivided into groups *Data* and *Acknowledge*.

The parameters of service primitives in a facility are related to each other. In the simplest case, the parameters of an indication or a confirm are identical to those of a request or response respectively. Similarly, the parameters of a response are directly related to those of the indication. However, more complicated possibilities exist. For quality of service negotiation, for example, the relevant parameter in the indication may be weaker than that in the request if the service provider cannot meet the request in full. The parameter in the response may again be weaker

than that in the indication if the responding user cannot meet the requirements in the indication. The parameter in a confirm is almost invariably the same as that in a response, but in general may vary. The specification of a facility should thus allow for a relation (that may not be identity) between an indication and a request, a response and an indication, a confirm and a response.

Service facilities may be invoked in an isolated fashion. This is the case for a connection-less service, for example, in which every data transfer is unrelated to others. Service facilities may also have some relationship to each other. This applies to a connection-oriented service, for example, in which connection establishment must precede data transfer and connection release. OSI uses the concepts of association, connection and connection endpoint to indicate that service facilities are related. However, this is not general enough since there are many possibilities between purely connection-less and connection-oriented.

The more general notion of an interaction group is therefore introduced in this paper. This is a collection of interactions that should be considered related. A service facility is an interaction group, and so are combinations of service facilities. Such a group needs a unique reference, called an interaction group identifier (IGId). Connections are interaction groups, and connection endpoint identifiers are interaction group identifiers. Interaction group identifiers are known only locally to a user, so a pair of identifiers is associated with one interaction group. Strictly speaking, an invocation (instance) of a service user deals with each interaction group. However, the term 'user' is used widely although 'service user invocation' would be the accurate description.

3.2.2 Patterns of Service Facility

A study of typical OSI services reveals that there are five common patterns of service facility. These are illustrated in figure 2 according to the service conventions of [1]. The service conventions document defines simple time-sequence diagrams in which time runs down the page, and three columns describe the interactions between two users and the service provider as intermediary. Arrows indicate occurrence of a service primitive, and sloping lines suggest the time delay between the occurrence of a service primitive at one user and the corresponding occurrence at the other. When the occurrence of two service primitives is not time-related, a tilde ($\sim$) is placed between them.

The time-orderings among primitive occurrences in a facility constitute a temporal constraint. Each basic pattern may have one of the properties illustrated in figure 3. The properties are arranged in a hierarchy:

single: a single occurrence of the facility is permitted, e.g. to initialise a service; otherwise, multiple occurrences are permitted

consecutive: multiple occurrences strictly follow each other, e.g. to ensure that an expedited data request is dealt with before another one is allowed; otherwise, overlapped occurrences are permitted

ordered: overlapped occurrences respect the relative order of primitives in different invocations of a facility, e.g. to ensure that acknowledged data transfer is properly pipelined; otherwise, overlapped occurrences are unordered with respect to each other

reliable: unordered occurrences are fully completed, e.g. to ensure that data transfer requests are not lost; otherwise, unordered occurrences are unreliable

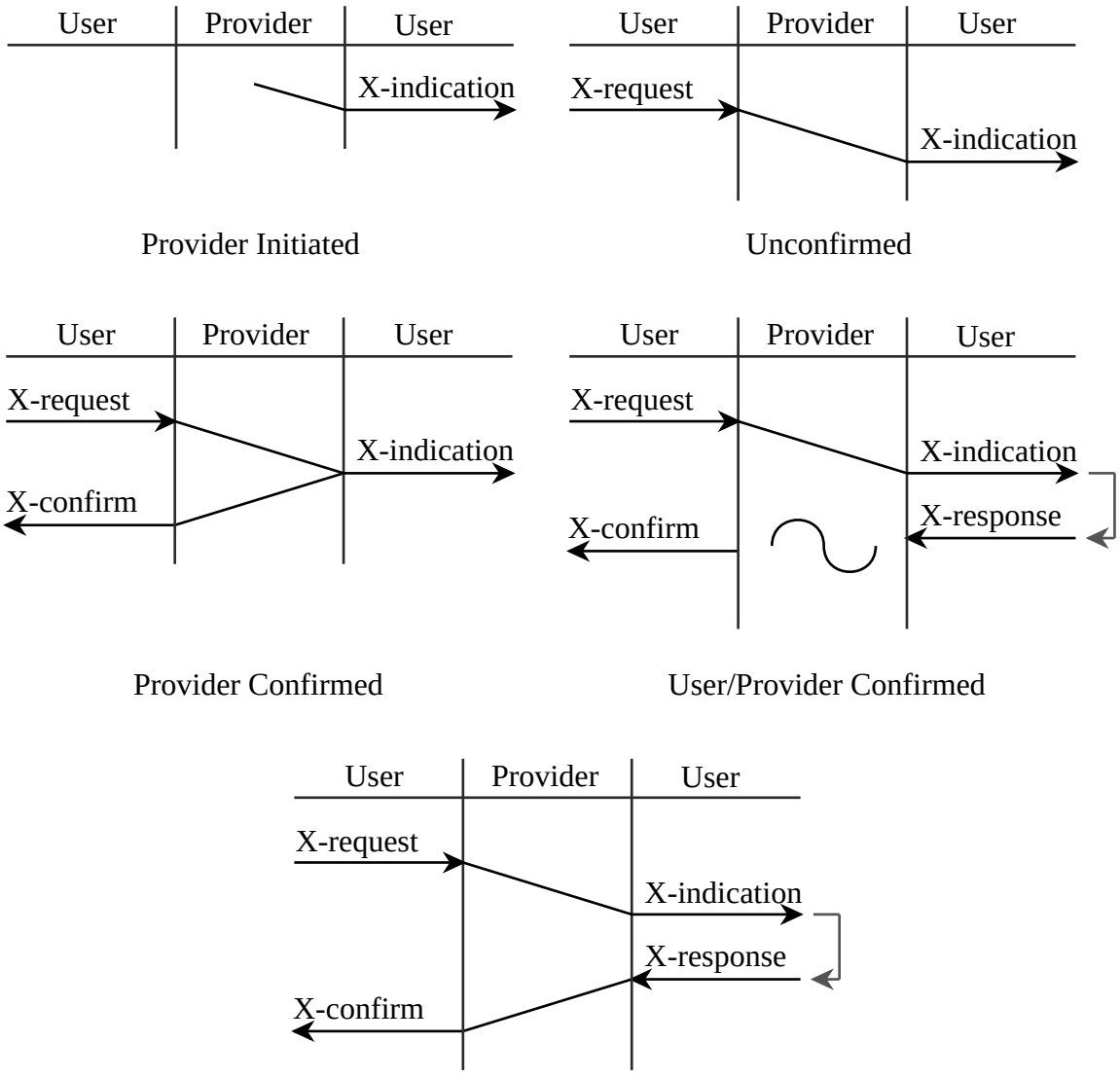

Figure 2. Service Facility Patterns

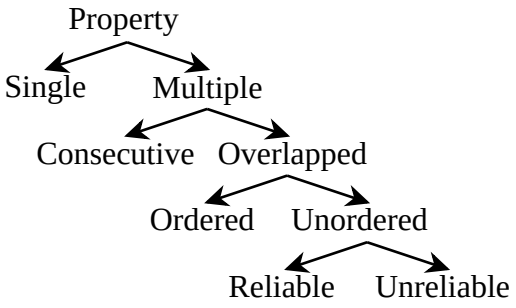

Figure 3. Service Facility Properties

unreliable: unordered occurrences may not be fully completed, e.g. if data requests may be lost due to problems in the service provider such as congestion.

Facilities also have a direction, relating a particular pair of users and therefore interaction group identifiers. Although many services are symmetrical, there may be asymmetries in what users can invoke. For example, some users may be allowed only to initiate connections while others are allowed only to respond to connections. Within a connection, only the initiator may be allowed to send data or only the responder may be allowed to break the connection. Facilities should thus be specified unsymmetrically, but a symmetrical service can simply allow facilities between all (distinct) pairs of users.

3.2.3 Service Facility Specification

Service facilities and their combinations will be described using the language SAGE (*Service Attribute Generator*). The language is briefly explained in this paper, but [9] should be consulted for more details. In particular, [9] gives semantics to the language by means of denotations in LOTOS for each type of declaration. There is insufficient space in this paper to show the LOTOS specifications that are generated from service declarations. However, the essence of SAGE is architectural; its semantics could in principle be given in other terms such as trace logic.

Basic facilities are described by means of the following declaration:

facility(*direction*, *pattern*, *property*, *group1*, *group2*)

The direction is *12* or *21*, depending on which of the users initiates the facility; user 1 is conventionally the left-hand user in a time-sequence diagram, user 2 is the right-hand user. The pattern is one of the five patterns found in figure 2, and the property is one of the leaf properties found in figure 3.

In the case of a provider initiated or unconfirmed pattern, only the first group is given. In the case of a confirmed pattern both groups are given. The two groups are normally identical and name the service facility. However, in a subdivided facility with group names like *Data* and *Acknowledge* they may be different. The groups must also give the service primitive parameters. By default, the parameters in a request and indication or response and confirm are required to be the same, and no relationship holds between the parameters of an indication and a response. The specifier must alter the generated specification if a more complex relationship holds.

Here are some sample declarations of facilities:

facility(12,provider_initiated,single,Start()): a single start indication with no parameter may be spontaneously given to user 2 by the service provider

facility(21,unconfirmed,ordered,Expedited(Data)): an unconfirmed expedited data facility with a data parameter may be sent by user 2 to user 1; a further request may be made before the previous one has been dealt with, but the order of transmission is respected

facility(12,user_confirmed,consecutive,Connect(Addr,Addr),Accept(Addr)): a user confirmed connect with two address parameters may be sent by user 1 to user 2, the accept response having one address; a further request may not be made until the previous one has been honoured.

Since facilities in each direction between users are the same if the service is symmetrical, a facility in the reverse direction between the same users may be declared with:

reverse(*facility*)

A range of basic service primitive parameter sorts are pre-defined, so that they may be used immediately in the declaration of facilities. The available types are:

Title, Addr: a service user title and service access point address, specified as distinct labels without structure

IGId, IGIdSet: interaction group identifiers, specified as distinct labels without structure

Data: a service data unit, specified as a string of octets

Orig: the originator of a facility, specified as being 'user', 'provider' or 'other'

Reas: a reason for invoking a facility, specified as distinct labels without structure

Opt: a service option (a functional aspect such as expedited data selection, or qualitative aspect such as throughput), specified as a generic type with comparison of option values (for negotiation)

Prim: a service primitive, defined using the information from the facility declarations

PrimQ: a queue of service primitives being processed, specified as a string of primitives

Other types may be used freely as service primitive parameters, but their formal definitions must be added by the specifier. The definitions of the above types may also need to be modified or replaced by the specifier. For example, a specific address structure may be needed or specific options may be defined.

3.3 Service Combinations

3.3.1 Service Combinators

In principle, service facilities could be combined in a limitless number of ways. However, OSI standards typically use a small number of common combinations. These are discussed below along with how they are declared in SAGE. The declarations of combinations give one or two behaviours to be combined. The behaviours are those of basic facilities or their combinations. A facility group is given as parameter when it is necessary to qualify the combined behaviour as applying to a particular facility within it. In the following declarations, the italicised behaviours would be defined directly as facilities or using other combinators. Each declaration takes the form:

combinator(*parameter_1*,. . . , *parameter_N*)

Such a declaration stands for the behaviour given by its parameters, combined in a particular way. Combinators may therefore be built up into larger expressions such as:

combinator_1(*combinator_2*(...),*combinator_3*(...))

Sometimes a single large expression for a service would be unwieldy, or would require repetition of sub-expressions. In such a case, a part of the overall behaviour may be defined by:

define(*behaviour,combinator*(...))

where *behaviour* would be used as a parameter to other combinators. Typically this is useful for giving a name to the behaviour of each service facility.

3.3.2 Enabling and Disabling

The completion of one facility may allow another behaviour to start. For example, completion of service selection may be necessary before it can be used:

enables(*selection,usage*)

One facility may be able to interrupt and terminate another. For example, disconnection may disable data transfer:

disables(*disconnect,data*)

The *disables* combinator causes permanent disruption. Instead, a facility may be interrupted but then resumed after completion of the interrupting request. For example, reset interrupts data transfer but allows it to continue (with a fresh start) after the reset:

interrupts(*reset,data*)

Although enabling is an obvious relationship between two facilities, it does not usually appear in a service in quite this form. A more normal situation is that the facility is enabled for each user separately after local completion. Consider user-confirmed connection followed by data transfer. After the connect response, the responding user may immediately begin data requests even though the connect confirm has not yet been delivered to the initiating user. (The connect confirm will, of course, occur before the corresponding data indications.) After the connect confirm, the initiating user may begin local data requests. This behaviour is so common that a special declaration is available for it. The name comes from the fact that one behaviour may enable another immediately after an acknowledgement (a response or confirm). Connection enabling data transfer might thus be declared by:

enables_after_ack(*connection,data*)

Another variation of enabling occurs when a facility is allowed to begin as soon as another has been initiated. In this case, it is the request or indication that enables the facility locally. Typically, this arises for disconnection. A disconnect makes no sense until a connection has been attempted, but may be requested before a connection has been confirmed; this allows either user to abandon a connection attempt. It is the try rather than acknowledgement of connection that allows disconnection to take place. Connection enabling disconnection is thus declared by:

enables_after_try(*connection,disconnection*)

In all three variations of enabling, the first facility enables the second and then ceases to operate. A common requirement is for the whole combination to repeat after the second terminates. Connection followed by disconnection is a particular example, since completion of disconnection allows a new connection attempt to begin. This differs from the case of enabling in that cyclic (recursive) behaviour is possible. There are therefore two variants of the *interrupts* combinator, used according to whether the second facility must reach the stage of acknowledgement or just trying:

interrupts_after_ack(*disconnection,connection*)
interrupts_after_try(*disconnection,connection*)

3.3.3 Duplexity

Two facilities may be entirely independent. For example, data transfer in each direction between a pair of users is usually separate and may be declared by:

interleaves(*data12,data21*)

A facility may be used alternately by each user. For example, data transfer in each direction may be two-way alternate (‘half duplex’), declared by:

alternate(*data*)

Instead of this, a facility may be used at the same time by both users. Thus, for two-way simultaneous (‘full duplex’) transfer of expedited data the declaration would be:

simultaneous(*expedited*)

3.3.4 Interference

One facility may have priority over another, such that its requests may be dealt with first. For example, the following declaration says that expedited data may overtake normal data (although this is not guaranteed):

overtakes(*expedited,normal*)

If the same facility is invoked ‘simultaneously’ by both users, a collision of requests will occur inside the service provider. For some facilities (such as data transfer), the requests will not interfere with each other. For others (such as disconnection), the requests are mutually supportive. In such a case, only some of the primitives of the facility occur: for an unconfirmed facility, there are requests only; for a confirmed facility, there are requests and confirms only. For example, the collision of disconnects may be declared with:

colliding(*disconnect*)

3.3.5 Global Aspects

The combinators seen so far deal with pairs of specific users. There are various ways in which such behaviours can be combined for all of them. The following declaration says that the behaviour applies to all distinct pairs of users (strictly, interaction group identifiers):

forall_ids(*behaviour*)

Each user must use distinct interaction group identifiers, though the same value might be used concurrently by several users. Some facility must start an interaction group (causing its identifier to be allocated). Some facility (possibly the same one) must end an interaction group (causing its identifier to be de-allocated). For example, to declare that connection and disconnection play this role:

unique_ids(*connection,disconnection,behaviour*)

Note that *connection* and *disconnection* here are facility groups and not behaviours.

At a global level, the service provider may temporarily withhold the opportunity to invoke certain kinds of facility. This might apply to connection or data transfer, for example, due to congestion within the service. The effect is that some (perhaps all) users are prevented from issuing certain requests for a time. Consider backpressure flow control, which withholds data requests until the data pipeline is sufficiently clear. This would be declared as:

withheld(*data,behaviour*)

Note that *data* here is a facility group and not a behaviour.

Finally, the ultimate composite behaviour of a service must be declared as the global one. At the same time, a name for the kind of service is declared. For example, a connection-oriented service might be declared as:

global(*co,behaviour*)

3.4 Example Service Declarations

A basic connection-less service has the following characteristics. A datagram facility allows unrelated data messages to be sent by one user to any other. Multiple data transfers may be initiated by a user; these may be overlapped, may arrive in a different order, and may not be reliably delivered. Datagrams have a source address, a destination address, and a data parameter. This service is represented by the following declarations:

```
define(datagram,
  facility(12,unconfirmed,unreliable,Datagram(Addr,Addr,Data)))

global(cl,forall_ids(datagram))
```

An acknowledged connection-less service is like a basic one, except that datagram arrival is confirmed. Supposing that the service provider confirmed delivery and guaranteed reliable transfer, the declarations would be:

```
define(datagram,
  facility(12,provider_confirmed,reliable,Datagram(Addr,Addr,Data)))

global(acl,forall_ids(datagram))
```

It is possible to describe unsymmetrical services; indeed these are perhaps more complex and therefore a greater test of the expressive power of SAGE. The next (somewhat extreme) example uses an unsymmetrical connection-oriented service with the following characteristics. A user-confirmed, reliable connection facility allows connections to be established between a pair of users; the addresses of the initiating and responding users are provided as parameters when connection is tried. The connection facility may be temporarily withheld from some users. Once a connection has been tried, it may be broken by disconnection; a connection may then be tried again. Once a connection has been acknowledged it is possible to invoke normal data, expedited data and reset facilities. A bidirectional, provider-confirmed, unreliable data facility allows normal data to be transferred by either user; a data parameter is provided when data transfer is tried, and an acknowledgement is returned on successful delivery. An unconfirmed, reliable expedited data facility allows only the responding user to send priority data; this carries a data parameter. A colliding, unconfirmed, reliable reset facility allows data transfer to be interrupted and resumed from scratch; a reason parameter is supplied when reset is invoked. An unconfirmed, reliable disconnect facility allows only the responding user to break a connection. This service is represented by the following declarations:

```
define(conn,facility(12,user_confirmed,reliable,Conn(Addr,Addr),Conn))
define(norm,facility(12,provider_confirmed,unreliable,Data(Data),Ack))
define(exp,facility(21,unconfirmed,reliable,Exp(Data)))
define(reset,facility(12,unconfirmed,reliable,Reset(Reason)))
define(disc,facility(21,unconfirmed,consecutive,Disc))

global(co,
   withheld(Conn,
      unique_ids(Conn,Disc,
         forall_ids(
               interrupts_after_try(disc,
                  enables_after_ack(conn,
                     interrupts(colliding(reset),
                        interleaves(norm,
                           overtakes(exp,reverse(norm)))))))))
```

Note that group names have been capitalised here (e.g. *Conn*) to distinguish them from facility names (e.g. *conn*).

3.5 Tool Support

The SAGE language has been implemented as a library of macros written in the *m4* language. The macros define the language by producing LOTOS text for each declaration. The overall shape of the specification, data type definitions and process definitions are generated by the macros. The library contains about 80 macros in 1400 lines of *m4*. Most of the macros are auxiliary, to support the declarations given in SAGE. Once a specification has been generated automatically, the specifier may modify it to deal with finer points that are not handled by SAGE. For example, the specifier might introduce specific address formats, specific quality of service parameters, and specific constraints on quality of service negotiation. [9] gives fuller details of the translation process and the LOTOS generated for each service declaration.

4 Engineering Digital Logic

4.1 Digital Logic

It has been shown how communications services can be formally engineered in terms of their components and combinations. This example is rather high level, and uses constraints to express the operation of a service. As a contrasting example, it will now be shown how digital logic designs can be formally engineered using models of hardware components and their combinations.

Digital logic design is much better understood than service engineering; many textbooks explain the operation of logic gates and how to combine them into larger circuits. Furthermore, digital logic design is in practice constrained by the availability of specific hardware components that might be found in any manufacturer's catalogue. Although many components might in principle be chosen for building digital logic, a component-engineering style should be grounded in reality. This allows standard components and combinations to be used, and ensures a clear relationship between this approach and standard logic design.

Hardware specification has been extensively investigated. Languages such as CIRCAL (*Circuit Calculus*), HOL (*Higher Order Logic*), RTL (*Register Transfer Language*), VHDL (*VLSI Hardware Description Language*) and many others have been used to specify and analyse hardware. A component-engineering style for formal design of digital logic is therefore well-accepted. In common with all such approaches, the goal of the work reported in this paper is to allow digital logic designs to be specified, analysed and verified before actually building hardware. However, the emphasis here is to identify clearly the components and their means of combination.

As with communications services, a language could be specially devised to support digital logic design. However, investigation has shown that LOTOS provides good support for digital logic design. Specifications will therefore be written directly in LOTOS, although a library of components has been developed to allow specifications to be written more easily. More details of the DILL[1] (*Digital Logic in LOTOS*) approach are given in [11]. A further goal of this work was to investigate the suitability of LOTOS for specifications in this application area.

4.2 Digital Logic Components

4.2.1 Modelling Digital Signals and Gates

Logic functions (logic gates) are the basic components of digital logic. They operate on binary-valued digital signals. It turns out that the way in which signals are modelled and handled is critical to the success of specifying digital logic in LOTOS. An inappropriate model results in obscure or unusable specifications. Some of the critical issues are discussed below.

In reality, signals take on a range of analogue values (e.g. from 0 to 5 volts) but thresholds are set so that signals may be treated as logic 0 or 1. As a signal changes from one value to another, it passes through an indeterminate state that is neither logic 0 nor 1. It might therefore seem that tri-state logic should be used, with the addition of an 'undefined' state for signals. This, however, would make specifications much more complex. An undefined state should always be transient and therefore should be ignored. As a workable abstraction, therefore, signals are regarded as having only two states called 0 and 1.

[1] The approach was developed by the author, in conjunction with Richard O. Sinnott who carried out the detailed specification and verification work.

There is also a choice of whether a signal level or a change in signal level should be modelled as a LOTOS event. Choosing to model signal levels means that a gate must repeatedly offer its current output value in events. This clutters the behaviour with identical repeated events. Events therefore correspond to establishment of a new level. This means, for example, that if the inputs to a gate change but the output stays the same, then there will be no new offer of an output event.

Gates must not insist on outputting a new value after an input changes. In circuits involving feedback (e.g. a flip-flop), this can lead to deadlock. In practice as well, there may be a short input pulse to which a gate cannot react quickly enough. Real gates have a propagation delay between an input change and the corresponding output; an input pulse of rather shorter duration may not produce an output. Allowing a further input before output is therefore both realistic and necessary.

Open circuits are possible in actual hardware. For example, an input may be left floating and an unused output may not be attached to anything. There is also a switch-on problem in that when a gate is powered on it needs a short time to stabilise. The solution is to parameterise each gate with the default values of its inputs. At switch-on, and for a floating input, these defaults apply. Subsequently the gate may receive actual values at its inputs which will replace the defaults. Floating outputs still produce values, but they go nowhere. In LOTOS terms, these are hidden internal events.

LOTOS offers more possibilities for dealing with inputs and outputs than are used in practice. An obvious solution is to make each input and output correspond to a LOTOS gate. This might be termed 'physical multiplexing', because each LOTOS gate corresponds to a physical port. LOTOS also allows what might be described as 'logical multiplexing', in which there would be one LOTOS gate that is qualified by a port number parameter in events. The advantage of this style is that a LOTOS gate may then correspond to arbitrary numbers of inputs or outputs. This does not faithfully reflect real logic gates, which are always built with a fixed number of inputs and outputs. Also it considerably complicates how the wiring up of components is specified. Physical multiplexing is therefore used.

Although gates with more than two inputs are perfectly possible, four and eight inputs tend to be the only other varieties found. Unused inputs can be wired to logic 0 or 1 as required to make them ineffective. LOTOS could allow a parameterised number of inputs by making use of logical multiplexing, but this would be too far removed from reality. A fixed number of inputs is therefore specified.

Real gates are connected by wires from outputs to inputs. The wires (should) accurately transmit signals, but they can introduce a propagation delay that is critical in high-speed circuits. The wires could be considered as components as well, but to do so would make the logic specifications very unwieldy. In virtually all logic designs the wires can be ignored, but where their effect is significant then they can be specified as delays. Ignoring the wires makes connection of components very easy in LOTOS: events at the relevant output and input gates are allowed to synchronise by giving them the same gate name. In effect, a gate name is given to a wire. Multi-way synchronisation in LOTOS also allows one output to be sent to several inputs[2].

[2]Trying to synchronise two outputs in LOTOS could well lead to deadlock. Trying to connect the outputs of two physical logic gates could lead to a more serious form of deadlock!

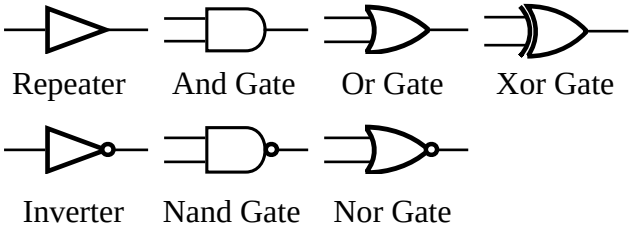

Figure 4. Logic Gate Symbols

4.2.2 Basic Logic Gates

The conventional symbols for the logic gates supported are given in figure 4. A one-input gate can perform one of two different logic functions: as a repeater (or amplifier or delay) and as an inverter. A two-input gate can perform one of 16 different logic functions. Only some of these are usually given names such as *and*, *or* (inclusive or), and *xor* (exclusive or). Certain logic functions are easier to implement in hardware, so *nand* and *nor* are also common.

Some logic gates could be built from simpler combinations. For example, a *nand* gate could be built from an *and* gate feeding into an inverter. The gate might actually be built this way, but the availability of *nand* gates in practice means that it is reasonable to specify them directly. An *and* gate with one input inverted is not, however, a normal hardware component so it would be specified as an inverter feeding into an *and* gate.

Hardware gates are designed to implement a fixed function; a ULA (*Uncommitted Logic Array*), PLA (*Programmable Logic Array*) or CLA (*Configurable Logic Array*) might be considered as an exception. LOTOS is more flexible in terms of parameterising a gate with its function. Although each kind of gate could be explicitly specified with its function, this would lead to a lot of duplication in specifications since the behaviour of a gate is largely separate from its actual logic function. The specification style therefore breaks from a strict representation of real gates by specifying a generic gate with its logic function as a parameter. Because LOTOS does not allow operations to be given as parameters to processes, the names of the operations rather than the operations themselves are given as parameters. An *Apply* operation takes an operation name and parameters, and calculates the results of the logic function. The specific operations supported are:

unary: *same* (for a repeater) and *not* (for an inverter)

binary: *and*, *nand*, *or*, *nor*, *xor*.

Names could be given to the other binary operations, but would rarely be needed and would be unlikely to correspond to actual gates. Ternary and higher operations could also be given specific names (e.g. for a four-way *and*) but are specified for simplicity using the binary operations.

Sometimes it is necessary to tie an input to logic 0 or 1. This is a nullary logic function, specified by a behaviour that outputs its parameter as a constant value:

```
process Constant [op] (bop : Bit) : noexit :=
   op ! bop;  stop
endproc (* Constant *)
```

The earlier discussion about how to model digital signals and gates leads to a surprisingly complex specification of a one-input, one-output logic gate:

```
process Logic1 [ip, op] (bop : BitOp) : noexit :=
  let b : Bit = 0 in
    op ! Apply (bop, b);  Logic1A [ip, op] (bop, b)
  []
    Logic1A [ip, op] (bop, b)
  where
  process Logic1A [ip, op] (bop : BitOp, b : Bit) : noexit :=
    let bold : Bit = Apply (bop, b) in
      ip ? b : Bit;  Logic1B [ip, op] (bop, b, bold)
  endproc (* Logic1A *)
  process Logic1B [ip, op] (bop : BitOp, b, bold : Bit) : noexit :=
    let bnew : Bit = Apply (bop, b) in
      [bnew ne bold] ->
        op ! bnew;  Logic1A [ip, op] (bop, b)
    []
        Logic1A [ip, op] (bop, b)
  endproc (* Logic1B *)
endproc (* Logic1 *)
```

The gate above is parameterised by a unary logic function. Initially it may output a result based on its default input value of 0, and then deal with input. Alternatively, it may input a new value and then produce an output if this has changed; this behaviour is repeated. As discussed in [11], considerable investigation was necessary in order to come up with this specification of a one-input gate. There are subtleties hinted at earlier which make it hard to specify logic components that assemble properly into high-level designs. Space does not allow a full discussion here of alternative specifications that are unsuitable.

As an example of a one-input logic gate, an inverter has the specification:

```
process Inverter [ip, op] : noexit :=
  Logic1 [ip, op] (not)
endproc (* Inverter *)
```

A two-input gate is specified much as a one-input gate, and is parameterised with the name of a binary logic function:

```
process Logic2 [ip1, ip2, op] (bop : BitOp) : noexit :=
  let b1 : Bit = 0, b2 : Bit = 0 in
    op ! Apply (bop, b1, b2);  Logic2A [ip1, ip2, op] (bop, b1, b2)
  []
    Logic2A [ip1, ip2, op] (bop, b1, b2)
  where
  process Logic2A [ip1, ip2, op] (bop : BitOp, b1, b2 : Bit) : noexit :=
    let bold : Bit = Apply (bop, b1, b2) in
      ip1 ? b1 : Bit;  Logic2B [ip1, ip2, op] (bop, b1, b2, bold)
    []
```

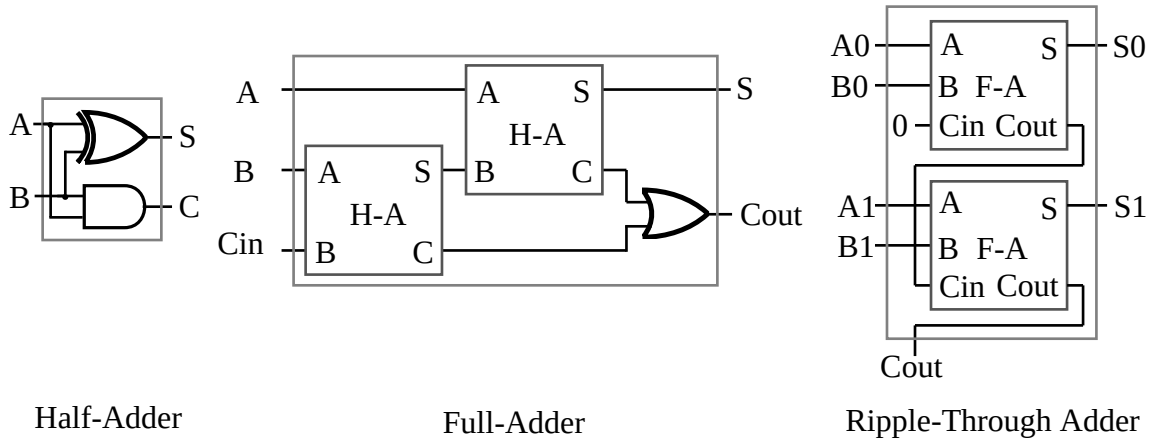

Figure 5. Examples of Adders

```
        ip2 ? b2 : Bit;   Logic2B [ip1, ip2, op] (bop, b1, b2, bold)
    endproc (* Logic2A *)
    process Logic2B [ip1, ip2, op] (bop : BitOp, b1, b2, bold : Bit) : noexit :=
      let bnew : Bit = Apply (bop, b1, b2) in
        [bnew ne bold] ->
          op ! bnew;   Logic2A [ip1, ip2, op] (bop, b1, b2)
      []
          Logic2A [ip1, ip2, op] (bop, b1, b2)
    endproc (* Logic2B *)
endproc (* Logic2 *)
```

As an example of a two-input gate, a *nor* gate has the specification:

```
process Nor2 [ip1, ip2, op] : noexit :=
  Logic2 [ip1, ip2, op] (nor)
endproc (* Nor2 *)
```

4.3 Digital Logic Combinations

Logic gate components are combined according to standard patterns for circuits. These may be found in any reference on digital design such as [4]. Combinations are therefore given to the specifier; the requirement is to represent these easily in LOTOS. Two kinds of circuit are used below as illustration: adders and flip-flops. It should be noted from the examples how easily simpler components can be combined into larger ones.

4.3.1 Adders

Adders perform bit-by-bit additions on binary numbers. The design of some common kinds is shown in figure 5. There are other kinds of adder and arithmetic unit that will not be discussed here.

A half-adder produces a sum S and carry C from two binary inputs A and B, using an *xor* gate for the sum and an *and* gate for the carry. Its LOTOS specification directly mirrors its design:

```
process HalfAdder [A, B, S, C] : noexit :=
   Xor2 [A, B, S]  |[A, B]|  And2 [A, B, C]
endproc (* HalfAdder *)
```

A full adder also takes a carry resulting from the addition of a previous pair of bits. It therefore has both carry in and carry out, Cin and $Cout$. In this and later examples, hidden LOTOS gates are introduced to carry internal signals. The construction and specification of a full adder require two half-adders and an *or* gate:

```
process FullAdder [A, B, Cin, S, Cout] : noexit :=
   hide Sint, Cint0, Cint1 in
      (HalfAdder [A, Sint, S, Cint0] |[Sint]| HalfAdder [B, Cin, Sint, Cint1])
   |[Cint0, Cint1]|
      Or2 [Cint0, Cint1, Cout]
endproc (* FullAdder *)
```

A ripple-through adder adds pairs of bits in parallel, but the carry must ripple through from earlier additions to later ones before the output is stable. The number of bits to be added must be fixed, so a two-bit adder has been chosen for concreteness. However, the idea works for an arbitrary number of bits, with a full adder for each pair of bits. Since there is no initial carry to the adder, the first carry input is tied to 0.

```
process RippleThroughAdder2 [A0, B0, A1, B1, S0, S1, Cout] : noexit :=
   hide Cint0, Cint1 in
      (Constant [Cint0] (0)  |[Cint0]|  FullAdder [A0, B0, Cint0, S0, Cint1])
   |[Cint1]|
      FullAdder [A1, B1, Cint1, S1, Cout]
endproc (* RippleThroughAdder2 *)
```

4.3.2 Latches and Flip-Flops

Latches and flip-flops are bistable devices. The design of some common kinds is shown in figure 6. There are other kinds of latches and flip-flops that will not be discussed here.

An RS latch is named after its R (Reset) and S (Set) inputs. There are two outputs: the standard output, conventionally named Q, and its negation, $\overline{Q}$. Resetting the latch causes Q to become 0 and $\overline{Q}$ to become 1; setting does the opposite. An RS latch can be built from two cross-coupled *nor* gates. Its specification in LOTOS is a straightforward reflection of the standard design:

```
process RSLatch [R, S, Q, Qbar] : noexit :=
   Nor2 [R, Qbar, Q]  |[Q, Qbar]|  Nor2 [S, Q, Qbar]
endproc (* RSLatch *)
```

The RS latch may be set at any time by changes in its inputs. This may be undesirable if there is a risk of fluctuations in the inputs or if synchronous logic is required. A clocked RS latch may therefore be built out of a basic RS latch. This has an additional clock input, C. The clock input must be 1 before resetting or setting will have any effect. The LOTOS specification of this uses two *and* gates and an RS latch in the standard way:

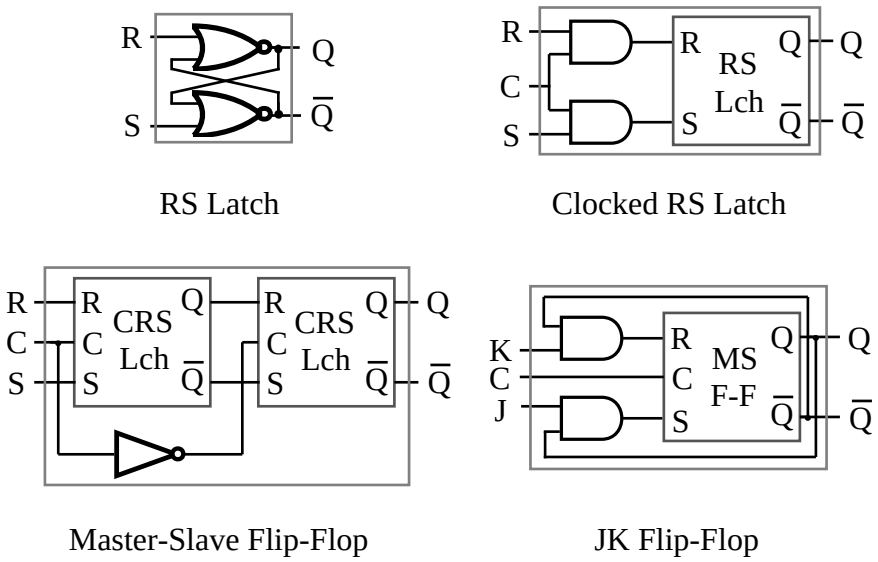

Figure 6. Examples of Latches and Flip-Flops

```
process CRSLatch [R, S, C, Q, Qbar] : noexit :=
  hide Rint, Sint in
    (And2 [R, C, Rint]  |[C]|  And2 [S, C, Sint])
  |[Rint, Sint]|
    RSLatch [Rint, Sint, Q, Qbar]
endproc (* CRSLatch *)
```

RS latches can suffer from pulsing problems and race conditions when combined. These can be addressed by cascading two clocked RS latches in a configuration called an MS (*Master-Slave*) flip-flop. When the clock signal becomes 1, the master may be reset or set. When the clock signal becomes 0, the master can no longer be reset or set, and its state is transferred safely to the slave. The specification of this flip-flop in LOTOS combines two clocked RS latches with an inverter:

```
process MSFlipFlop [R, S, C, Q, Qbar] : noexit :=
  hide Rint, Sint, Cint in
    Inverter [C, Cint]
  |[C, Cint]|
    (
      CRSLatch [R, S, C, Rint, Sint]
    |[Sint, Rint]|
      CRSLatch [Rint, Sint, Cint, Q, Qbar]
    )
endproc (* MSFlipFlop *)
```

An MS flip-flop is still not robust enough to be used as a memory element, since it allows setting and resetting at the same time; this may lead to an indeterminate state. The final design to be considered is the JK flip-flop which avoids this problem by gating the inputs with the

opposite current output. The inputs to this kind of flip-flop are conventionally called J and K. The specification, like the design, requires two *and* gates in addition to an MS flip-flop:

```
process JKFlipFlop [K, J, C, Q, Qbar] : noexit :=
  hide Rint, Sint in
    (And2 [K, Qbar, Rint] ||| And2 [J, Q, Sint])
  |[Q, Qbar, Rint, Sint]|
    MSFlipFlop [Rint, Sint, C, Q, Qbar]
endproc (* JKFlipFlop *)
```

4.4 Tool Support

The DILL approach is supported by a library of macros written in the *m4* language. The macros are merely a convenient means of parameterising and generating LOTOS text for each kind of component or combination. Process definitions are generated by the macros for the components required in the design. The library contains about 40 macros in 800 lines of *m4*. Fuller details of the component and design library are given in [6, 11]. The specification of every individual component in the library has been checked in considerable detail with tools, although not yet formally verified. The ultimate objective is to have a fully verified library that can be used with confidence in designs of larger logic systems.

5 Conclusions

The place of formal methods in computing has been discussed. Scientific aspects of formal methods are dealt with in theoretical computer science. Engineering aspects are dealt with in formal software engineering and formal hardware engineering. Of these, formal software engineering is at a comparatively early stage and requires much more effort. Some of the issues needing attention include closer alignment with industrial needs, more case studies, relevant development models and metrics, professional recognition and management education.

The key aspect of success in engineering has been suggested to be use of known components, combined in known ways, yielding predictable results. This philosophy is also applicable to engineering with formal methods. The idea has been illustrated with two rather disparate application areas: developing high-level specifications of communications services, and developing low-level specifications of digital logic designs. Space has not allowed the full details to be explained, but they are documented separately for the interested reader.

A component-based style is believed to be generally applicable. The author and his colleagues have made preliminary investigations of the idea in other areas such as communications protocols, distributed systems and artificial neural networks. By acting as a practical aid to formal specification and design, a component-based style has some claim to being an engineering approach to formal methods.

Acknowledgements

The author is grateful for Richard O. Sinnott, University of Stirling, for the detailed specification and validation of digital logic designs in LOTOS.

References

1. ISO: *Information Processing Systems – Open Systems Interconnection – Service Conventions*, ISO TR 8509, International Organisation for Standardisation, Geneva, 1989.
2. ISO: *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, ISO 8807, International Organisation for Standardisation, Geneva, 1989.
3. ISO: *Information Processing Systems — Open Systems Interconnection — Guidelines for the Application of ESTELLE, LOTOS and SDL*, ISO TR 10167, International Organisation for Standardisation, Geneva, 1991.
4. Kline, R. M.: *Structured Digital Design*, Prentice-Hall, 1983.
5. Merriam-Webster: *Ninth Collegiate Dictionary*, Webster, 1988.
6. Sinnott, R. O.: *The Formally Specifying in LOTOS of Electronic Components*, M.Sc. Thesis, University of Stirling, 1993.
7. Turner, K. J.: 'An Architectural Semantics for LOTOS', Proc. 7th International Conf. on *Protocol Specification, Testing, and Verification*, pp. 15–28, 1987.
8. Turner, K. J. *(ed.)*: *Using Formal Description Techniques*, John Wiley, 1992.
9. Turner, K. J.: 'SAGE: A Language for Service Attribute Generation', 1993 *(in preparation)*.
10. Turner, K. J. and van Sinderen, M.: 'OSI Specification Style for LOTOS', Proc. 3rd LOTOSPHERE Workshop, Pisa, pp. 5/1–22, September 1992.
11. Turner, K. J., Sinnott, R. O.: 'DILL: Specifying Digital Logic in LOTOS', 1993 *(submitted for publication)*.