Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

Prioritised Service Architecture

Ian EAST
Dept. for Computing, Oxford Brookes University, Oxford OX33 1HX, England.
ireast@brookes.ac.uk

Abstract. Previously, Martin [1] gave formal conditions under which a simple design rule guarantees deadlock-freedom in a system with service (client-server) architecture. Both conditions and design rule may be statically verfied. Here, they are re-arranged to define *service protocol*, *service network* (system), and *service network component*, which together form a model for system abstraction. Adding mutual exclusion of service provision and dependency between service connections enriches abstraction and is shown to afford compositionality. *Prioritised alternation* of service provision further enriches abstraction while retaining deadlock-freedom and denying priority conflict, given appropriate new design rules.

1 Abstraction and Design

This work is predicated on the belief that abstraction is *paramount* in the engineering of any system. Abstraction must capture system behaviour and provide for reduction via design. *Compositionality* is therefore an essential consequence of the definition of system and component. Every system must form a valid component (for composition) and every component a valid system (for decomposition).

It is further held that *concurrency*, *priority*, and *alternation*, are essential forms of abstraction and their expression should be supported by a programming language. Concurrency is typically neglected because it invites pathological behaviour, such as deadlock, and the possibility of interference. However, formal *design rules* can provide security against most, possibly all, pathology [2, 3, 4]. Components, each using a different rule, may be composed without loss of guarantee, for example, of deadlock-freedom [5].

Guaranteeing interference-freedom in compositional concurrent system abstraction remains a challenge [6] and is not addressed here.

Design rules should be applied, and verified, as design proceeds. They thus need to form part of the model for system abstraction, to be incorporated within the language in which design is expressed, and to be *statically verified*. None of this is new. Structured Programming may be regarded as the abstraction of procedure, enforced by design rules expressed as the syntax of a ”high-level” programming language, such as Pascal. In return for the loss of some personal freedom in the way an algorithm is expressed, considerable security against error is gained.

occam extended this principle with the incorporation of “usage rules” – statically verified design rules which deny, for example, aliasing and concurrent access to a variable.

Static verification affords *correctness by construction*. The alternative is “trial-and-error”, which is *not* engineering. With static verification, every valid program is *a priori* guaranteed free of certain errors. Obtaining the same security by trial-and-error introduces unpredictable delay and requires additional capability in the development team. Both cost and risk increase.

Honeysuckle [7] relies upon a development of the Brinch-Hansen master-slave (client-server) protocol for all communication [8]. Peter Welch *et al.* first applied this to systems with

communicating process architecture, and provided a proof of deadlock-freedom [3]. Jeremy Martin later provided a formal foundation and proof using the failures model in CSP [1, 9].

Honeysuckle requires formal definitions that are compositional and which can be efficiently verified upon compilation. Its model for abstraction relies upon the notions of *service protocol* and *service network (component)* which will be adequately defined in the next section, giving rise to the notion of *service architecture*.

Honeysuckle also seeks to serve engineers of *reactive systems*, which respond to external events via pre-emption according to some prioritisation. Such needs are typically met at the hardware level by *prioritised vectored interrupts*. Normal control-flow may be interrupted by a signal, whereupon an interrupt service routine is executed. Behaviour may be said to *alternate* between processes. (This should not be confused with the ALT construct in occam. ALT might better abbreviate *alternative* and denotes a "one-off" selection.) Hoare described an alternation operator which relied upon an external signal, so that interruption was outside the control of either process, and which did not prioritise [10, #5.4]. The author has previously proposed a prioritising operator and programming construct. These are summarised in a companion paper which explores their semantics [11].

Section 3 extends the notion of service architecture to include prioritised alternation, thus introducing *prioritised service architecture*, showing how deadlock-freedom may be retained and priority conflict denied.

2 Service Architecture

2.1 Hierarchical Data-Flow

Brinch-Hansen introduced the idea of enforcing a hierarchical dependency between "masters" and "servants" in order to avoid deadlock between communicating processes [8]. He defined service as the receipt of a request, possibly followed by a reply, and began with a "basic assumption":

> A servant will always eventually receive a request, and (if required) send a reply, unless delayed indefinitely by one of its own servants.

A simple inductive proof is then given that deadlock can never occur.

Rather than label processes 'master' or 'servant' (or the more liberal 'client' or 'server'), an oriented arc drawn between the two should be labelled 'serves'. A system can be abstracted by a directed graph, where each node represents a process, and each arc represents service provision.

Design proceeds from higher to lower levels of abstraction. Thus some means must be found by which to guarantee the basic assumption. Brinch-Hansen lists four conditions, establishing a *protocol* between connected components.

While, as Brinch-Hansen points out, such a protocol may be implemented simply via stepwise refinement, there can be no guarantee that it is then always in force. Errors are possible. These might be detected by some additional verification tool. However, such a tool will not be easy to compose, and in any case affords only trial and error.

2.2 Service Protocol and Network

CSP affords a definition of service at a level of abstraction above data-flow. Furthermore, there is no need to be limited to a sequence of two communications. Provided all partners to service protocol are 'live' (never refuse *every* offer of communication) and *deterministic*

(allow the environment first authority to resolve any choice between two communications), the failures model [12] provides an adequate language in which to express the necessary conditions.

If, after trace s, an environment offers a process P the set of events X, and it refuses to participate in any of them, X is termed a *refusal* of P/s (P 'after' s). The combination of trace and refusal is referred to as a *failure* of P. Each failure is a very useful characteristic of a process as it constitutes a relation between past and imminent behaviour.

$$\mathit{failures}(P) = \{(s, X) \mid s \in \mathit{traces}(P),\ X \in \mathit{refusals}\,(P/s)\} \tag{1}$$

There is an important difference in the use of the terms 'client' and 'server'. Here, they refer to *ports* owned and regulated by the (distinct) processes that consume and provide a service. One should think of each end of a service connected to an appropriate port.

Definition 1 (Service). A *service* is a finite chain of communications between two processes P (provider) and Q (consumer)

$$S = \langle c_1, c_2, \ldots, c_n \rangle \ \subseteq\ \alpha P \cap \alpha Q \tag{2}$$

such that:

S1 *Client Condition.*
Q, as consumer, may request service at any time, by requesting c_1, but must then request each subsequent communication c_i immediately after the last, and continue to do so until it is granted.

$$\begin{aligned} \forall (s, X) \in \mathit{failures}(Q).\ \forall\, i \in N^{>}.& \\ s \downarrow c_1 \ =\ s \downarrow c_n \ &\Rightarrow\ \forall j \in N^{<}.\, c_j \in X, \\ s \downarrow c_i \ >\ s \downarrow c_{i+1} \ &\Rightarrow\ c_{i+1} \notin X \end{aligned} \tag{3}$$

S2 *Server Condition.*
P, as provider, may initially offer only c_1, and must eventually grant each subsequent communication until service completion.

$$\begin{aligned} \forall (s, X) \in \mathit{failures}(P).\ \forall\, i \in N^{>}.& \\ s \downarrow c_1 = s \downarrow c_n \ &\Rightarrow\ \forall j \in N^{<}.\, c_j \in X, \\ s \downarrow c_i > s \downarrow c_{i+1} \ &\Rightarrow\ \forall j \in N.\, j \neq i+1 \ \Rightarrow\ c_j \in X, \\ &\quad\ \exists (s \frown t, X') \in \mathit{failures}(P).\, c_{i+1} \notin X' \end{aligned} \tag{4}$$

where $N^{>} = \{1 \ldots n-1\}$, $N = \{1 \ldots n\}$, and $N^{<} = \{2 \ldots n\}$.

This definition suffices whether service is offered just once or continuously.

Service may be guaranteed to proceed, once initiated, and in strict sequence. Conflict-freedom is assured. No service may be simultaneously both available and in progress. Service is never 're-entrant', and is oriented, according to initiation *not* data-flow.

The applicable set of client ports – $clients(P)$ – and server ports – $servers(P)$ – may now be attributed to any process P. The interface between any pair of processes can then be recorded as follows:

$$interface(P, Q) = (clients(P) \cap servers(Q)) \ \cup\ (clients(Q) \cap servers(P)) \tag{5}$$

Each communication within a service must, at some stage, be qualified according to:

- orientation of data-flow
- whether value or object is passed
- type of value or object passed.

Note that an interface may now be expressed without reference to any *channel*. A service is a higher form of abstraction. Channels are needed only for the implementation of a service. At most two channels suffice (one in each direction of data flow).

Design may be restricted to systems comprising only processes which communicate entirely, and always, according to service protocol. A first attempt at a suitable definition, of both system and component, follows.

Definition 2 (Service Network (a)). A *service network* V is a set of concurrent processes such that:

S3 *Network Communication Condition*
Every communication forms part of some predefined service.

$$\forall P \in V.\ \forall c \neq \tau\ \in \alpha P.\ \exists S \in (servers(P) \cup clients(P))\,.\, c \in S \tag{6}$$

S4 *Network Composition Condition*
Every service provided/consumed by any component of V is either consumed/provided by another component or forms part of the system interface.

$$\begin{aligned} &\forall P \in V. \\ &\quad \forall S \in servers(P).\ (\exists Q \in V.\, S \in clients(Q)) \not\Leftrightarrow S \in servers(V), \\ &\quad \forall S \in clients(P).\ (\exists Q \in V.\, S \in servers(Q)) \not\Leftrightarrow S \in clients(V) \end{aligned} \tag{7}$$

S5 *Network Client Condition (a)*
No component of V may ever refuse everything.

$$\forall P \in V.\ \forall (s, X) \in failures(P).\, X \neq \Sigma \tag{8}$$

S6 *Network Server Condition (a)*
Every component of V must either offer all its services, be actively providing at least one, or offer none.

$$\begin{aligned} \forall P \in V.\ \forall (s, X) \in failures(P). \\ \forall S \in servers(P).\, c_1 \notin X \\ \vee\ \exists S \in servers(P).\ \exists i \in N^{<}.\, c_i \notin X \\ \vee\ \forall S \in servers(P).\, c_1 \in X \end{aligned} \tag{9}$$

As well as governing internal communication, S3 also implies that the *system* interface be composed entirely of services provided or consumed. Similarly, S4 implies that V itself has no 'loops' (connections between a client and server port within its interface), should it form a component of a larger system.

The above definition of a service network also defines a class of process, termed a *service network component*, or SNC. S5 ensures that no SNC may ever deadlock, livelock, or terminate.

S6 is entirely consistent with a SNC never offering *any* service. Such a component is termed *pure-client*. One that never *requests* any service is called *pure-server*. S6 prohibits a system/component offering just some of its services while none are in progress.

S5 has two additional consequences. Any pure-client SNC must, at all times, request or consume at least one service. This also holds for any component that is not pure-client, but which insists on withholding all its services (permitted by S6). Second, any SNC that requests services only in response to requests for those of its own must offer them all when not engaged in providing one. When so engaged, it may suspend the offer of any or all other services. This is simply the familiar "basic assumption" made by Brinch-Hansen. Whether a component is able to offer one service while providing another is left open.

System design is documented as a *service digraph*, which overlays the corresponding (undirected) communication graph. Each node represents a component process. Each arc represents a service provided, and is oriented from client to server.

It is quite a simple matter to compose systems whose service graph enjoys one or more directed circuits but which remain free of deadlock. Security must always be purchased with a certain loss of liberty. It is hereby proposed that circuit-free networks comprise a diversity sufficient to fulfil a worthwhile set of applicable specifications.

Conversely, given the above definition, it remains easy to contrive systems composed of valid SNCs which are not valid components themselves. Figure 1 depicts two such systems.

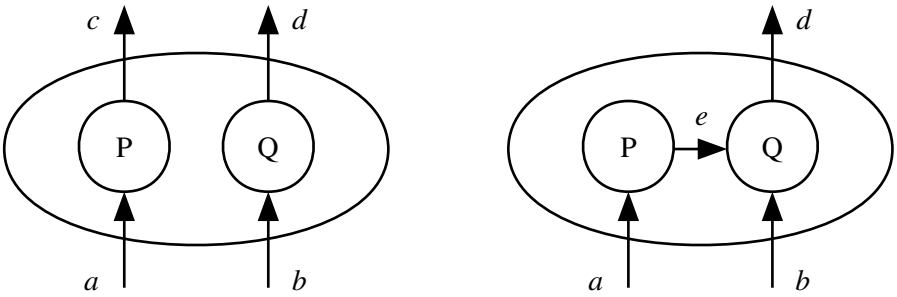

Figure 1: Service network components do not parallel-compose, under Definition 2.

On the LHS, services *a* and *b* are presumed dependent upon *c* and *d*, respectively. Following a request for *a*, the composite process is unable to satisfy any clause of S6. Once *P* requests *c*, it cannot immediately proceed with any service as provider. Neither can it offer all its services or deny them all.

On the RHS, *a* and *b* are presumed dependent upon *e* and *d*, respectively, but not *e* upon *d*. This time, following a request for *a*, S6 will be satisfied. *P* will withdraw *a*, and *Q* will withdraw *b*, when *e* is requested. However, following a request for *b*, *a* will remain on offer, while *b* suspends, pending completion of *d*. S6 is again denied.

It is the second clause of S6 that poses difficulty. Reliance upon it implies concurrency. To advance one service in progress, while offering others requires at least two processes. On the other hand, the set of SNCs, thus defined, does not close under the parallel operator. Two SNCs, composed in parallel, can together obtain states that breach the definition.

While a proof of deadlock-freedom is possible given Definition 2 and a simple design rule, it makes sense to seek compositionality first.

2.3 Composite Service Provision

With the rise in commercial importance of *component-based software development* [13], the need for precise and complete component specification is greater than ever. The ability to

freely interchange components, without denying the system specification, depends upon *compositionality*, which can be traced back to the 19th century philosopher Gottlob Frege [14].

> The meaning of a sentence must remain unchanged when a part of the sentence is replaced by an expression having the same meaning.

With reference to engineering, 'sentence' refers to the design of a system, where a more practical definition is needed. The term 'component' is preferred over 'part'.

Definition 3 (Compositional).
In order to recursively decompose a system into components, we require:

- some indivisible components
- that compositions of components are themselves valid components
- that behaviour of any system is manifest in its interface, without reference to any internal structure. Any such interface shall be termed *adequate*.

Components whose definition complies with all the above conditions may be termed *compositional*.

Corollary. It is then possible to substitute any component with another, possessing the same interface, without affecting either design or compliance with specification.

Corollary. Since the above definition must clearly apply recursively, closure is required in the definition of system and component. Every system should form a valid component and every component a valid system.

Jeremy Martin began by defining an indivisible component [1]:

Definition 4 (Basic Process). A *basic process* (BP) P is one that:

S1–3 communicates solely via service protocol

S4 possesses an interface comprising precisely those services it either provides or consumes

S5 may never refuse everything

S7 offers either all the services it is committed to providing, or none (satisfying S6).

$$\forall (s, X) \in failures(P).\\ (\forall S \in servers(P).\, c_1 \notin X) \ \vee (\forall S \in servers(P).\, c_1 \in X) \qquad (10)$$

It is obvious that any BP qualifies as a valid SNC, as thus far defined. Therefore any system composed entirely of BPs will be inherently deadlock-free. Note that the ability has been lost to offer other services while one is in progress, and that any BP may be expressed using only sequence, repetition (or recursion), and selection.

Martin termed a parallel composition of BPs a *parallel-composite process* (PCP). While any BP is a PCP, the converse is false. It is was thus necessary to separately prove that parallel compositions of PCPs are both deadlock-free and themselves compose. Worse, the new proof rests on the absence of any directed circuit in a new service graph where each node depicts a PCP. This denies many systems that would lack a circuit when reduced to the original graph, comprising only BPs.

An alternative approach was later proposed by Martin and Welch [9], whereby a network of PCPs is "exploded" (converted to one of BPs) and then tested for circuit-freedom. This does

not overcome the previous objection, and again reduces to trial-and-error, as the entire system must be re-assessed upon each refinement. (While a suitable programming environment might efficiently maintain an exploded description and rapidly add and test each refinement, such a description is *not* readily apparent to the designer, working at a (possibly much) higher level of abstraction.)

Clearly, a singular definition of system and component, together with a single design rule, is highly desirable, in order to permit secure recursive (de)composition.

2.4 Exclusion, Dependency, and Deadlock-Freedom

Services provided ("server connections", or just "servers") may be grouped into distinct *bunches*. Every server shall be a member of exactly one bunch.

$$\bigcup_i \mathit{bunch}_i(P) = \mathit{servers}(P), \qquad \bigcap_i \mathit{bunch}_i(P) = \emptyset \tag{11}$$

$$\forall\, S \in \mathit{servers}(P)\ \exists\, i.\ S \in \mathit{bunch}_i(P) \tag{12}$$

Within any bunch, services are now declared mutually *exclusive*. From the moment the delivery of any service starts until completion, no other member of its bunch may be offered. We shall also ensure that no service need be delayed by the delivery of one from another bunch. For example, two clients must queue to receive services within a common bunch. However, they may be served *concurrently* if the services they request each belong to a different one.

Any *dependency* between server and client connections is noted, indicating that the provision of one service requires the consumption of another. In the interface to any given component, any pair of client and server connections are either dependent or independent. When independent, the server need not be delayed by the client.

$$\mathit{dependencies}(P) \hat{=} \{(S, C)\}\ .\ S \in \mathit{servers}(P),\ C \in \mathit{clients}(P) \tag{13}$$

Any dependency applicable to a server connection is shared by every member of its bunch.

Note that the dependency set of any process does not emerge from its definition but forms part of that definition. In other words, one *specifies* the servers, clients, bunching, and dependencies, in order to define a system or component. Design and implementation follow.

Dependency forms a relation between server and client connections, characteristic of a particular component or system. It is transitive and forms a strict partial order on the arcs of a service digraph.

Exclusion and dependency affords the decomposition of a process interface into *server interface components*, each comprising a server bunch together with the set of clients on which it depends. Since there is also a sense in which these clients also 'depend' upon the servers, and because it makes a useful distinction, we shall refer to them as 'dependent' clients.

$$\begin{aligned} \mathit{SIC}_i(P) \quad = \quad & \{S \in \mathit{servers}(P) \mid S \in \mathit{bunch}_i(P)\} \ \cup \\ & \{C \in \mathit{clients}(P) \mid (S, C) \in \mathit{dependencies}(P)\} \end{aligned} \tag{14}$$

Other clients within an interface may be entirely *in*dependent. They pose no threat of deadlock since, by definition, they cannot compose to form a circuit of dependent services. Should they initiate a service, they remain bound by the service conditions (S1 and S2) and thus cannot give rise to conflict. It is useful to bunch any such ports together and refer to them as the *independent client component* of an interface.

Just as it was necessary to require the earlier form of service network (component) to be 'live' (never refuse all communication), a compositional service network must require all its server interface components to behave similarly.

Definition 5 (Service Network (b)). A *service network V* is a set of concurrent processes such that obey:

S3 *Network Communication Condition*

S4 *Network Composition Condition*

S8 *Network Client Condition (b):*
No component of *V* may ever refuse everything within any server interface component:

$$\forall P \in V.\ \forall i.\ \neg \exists (s, X) \in failures(P).\ SIC_i(P) \subseteq X \tag{15}$$

S9 *Network Server Condition (b):*
Every component of *V* must either offer either all services provided or none, within each server bunch (interface component):

$$\begin{aligned} &\forall P \in V.\ \forall i.\ \forall (s, X) \in failures(P). \\ &\qquad (\forall S \in bunch_i(P).\ c_1 \notin X)\ \vee\ (\forall S \in bunch_i(P).\ c_1 \in X) \end{aligned} \tag{16}$$

The definition of SNC is now closed under parallel composition. Unlike S6, S9 does not retain a clause that implies concurrency. It is also no longer possible to compose two SNCs in parallel to form something that is not itself a valid component.

Each interface component may be implemented using a single BP. Parallel composition of BPs introduces new interface components, and *vice versa*. S8 merely mirrors S7, which requires each BP to be 'live', but it allows an interface to be constructed without foreknowledge of implementation.

A design rule may now be stated and proved to guarantee deadlock-freedom:

Design Rule 1 (Service Architecture Design Rule). *(*SADR*)*
No circuit of dependent services is permitted in the service digraph.

Before we can offer a proof, a little more background is needed. For greater detail, refer either to the standard text [12] or Martin's thesis [1].

First, it is common to restrict interest to systems where communication occurs only between *pairs* of processes. Such systems are called *triple-disjoint*:

$$\forall \{i, j, k\}\ .i \neq j \neq k.\ \alpha P_i \cap \alpha P_j \cap \alpha P_k = \emptyset \tag{17}$$

When a process finds itself ready to engage in at least one communication (shared event), it is said to make a *request*. If its partner is unready or unwilling to comply, the request is said to be *ungranted*.

$$P_i \longrightarrow \bullet P_j\ \Leftrightarrow\ (\alpha P_i \cap \alpha P_j\ \subseteq\ X_i \cup X_j) \tag{18}$$

Each arc in a *snapshot digraph* represents an ungranted request. Any directed circuit manifest in a snapshot graph is termed a *cycle of ungranted requests* (CUR).

Conflict is said to occur between a pair of processes if both are ready to engage in a shared event but cannot agree on which one. Conflict-freedom implies the absence of any directed circuit of length two. Any circuit of length three or more is said to form a *proper cycle*, which is easily shown to be a necessary, but not sufficient, condition for deadlock.

A *system of interest* is one which is statically defined, connected, triple-disjoint, conflict-free, and composed of processes that are non-terminating and individually free of deadlock and divergence. None of these restrictions are severe in their consequences, and serve principally to simplify analysis.

The following theorem of Brookes and Roscoe can be used to prove deadlock-freedom in a wide range of systems, including service networks:

Theorem 1 (Fundamental Principle of Deadlock (FPD) [5]). *In any deadlock state of any system of interest, the snapshot digraph reveals at least one proper cycle of ungranted requests.*

Corollary. Any system of interest that never exhibits a proper cycle of ungranted requests is deadlock-free.

Corollary. Any system of interest whose communication graph contains no circuit is inherently deadlock-free.

There is no benefit here to separating the case of two processes in a "deadly embrace" from that of more than two. Conflict is a useful notion elsewhere, where local rules may be applied to pair-wise interaction. A generalization of the FPD [1, Theorem 1] may be applied instead that accounts directly for conflict-freedom, rather than treating it as a precondition. (It simply removes conflict-freedom from the definition of a system of interest.)

The following theorem may be proven by demonstrating that even when a CUR does occur, it is only temporary, and thus does indicate deadlock.

Theorem 2 (Service Network Theorem). *Any system whose service graph is free of any directed circuit of dependent services is free of deadlock.*

Proof. (adapted from that by Martin [1])

1. Assume a deadlock state exists. The FPD demands a CUR as a consequence.

2. Because there is no directed circuit of dependent services in the underlying service digraph, the CUR must contain a sub-path such that some process Q offers service to both its neighbours:

$$\exists P, Q, R \in V.\ \exists i, j \in N.\ P \xrightarrow{c_i} \bullet Q \xrightarrow{c_j} \bullet R \tag{19}$$

   P and R may represent the same process, which would then infer conflict.

3. The Client Condition (S1) allows R to refuse only the initiation of service, *i.e.* $j = 1$. (Whenever the orientations of ungranted request and service oppose, it is the service itself that is refused, *i.e.* its initial event.)

   Two possibilities must now be addressed separately: Either Q serves P and R via servers within a common bunch, or the two servers each lie in a separate bunch.

4. The Network Service Condition (S9) denies the possibility of a CUR passing through any single server bunch at Q. If $i = 1$ then c_i is granted. Q cannot offer service to R and not to P. If $i > 1$ then Q is denying P fresh service. It therefore cannot offer service to R.

5. Should the two servers at Q lie in separate bunches, a CUR through P, Q, and R is then possible. Another member of the bunch connected to P may be in progress, denying it service ($i = 1$). Should service be underway ($i > 1$), Q may be requesting or receiving a dependent service, and thus unable to immediately continue. Both situations

are temporary. The Server Condition (S2) guarantees that any service underway must eventually continue. The Network Client Condition (S8) ensures that any request for service will eventually be granted, provided there is no deadlock at a given server bunch (proven at previous step), and no "infinite overtaking" by other members (required in implementation).

There can thus be no CUR that corresponds to a deadlock state and therefore no deadlock. □

It is worth adding that S8 and S9 prevent not only deadlock but also infinite overtaking (and thus indefinite postponement) as a result of any preference a process might otherwise have for one interface component over another. However, such "unfairness" is still possible *within* any server bunch. As with any form of selection, resolution must be sought in implementation, which in practice is not usually a problem.

Note also that Step 5 also guarantees that a circuit in the underlying service digraph linking *independent* services can cause no deadlock either.

2.5 Summary

In place of the conditions listed *ad hoc* by Martin [1], there is now clear and separate formal abstraction of service and service network. The channel has been removed from system abstraction, as has the restriction on service protocol to sequences of just one or two communications. The definition of service network has been rendered compositional under the parallel operator by the introduction of exclusion and dependency – both natural forms of abstraction.

It remains to add priority and alternation to complete the arsenal, without compromising security.

3 Prioritised Service Architecture

3.1 Prioritised Alternation

Many applications require systems that respond to events that occur in their environment according to some prioritisation. Such systems are said to exhibit *reactive* or *event-driven* behaviour. Processor architecture often provides for this with *prioritised vectored interrupts*. Some external mechanism provides both an interrupt signal and an interrupt vector. The signal causes interruption of normal (sequential) control flow. The vector identifies the interrupt service routine subsequently executed. Upon completion of that routine, control returns to the interrupted process. Resumption must typically await completion, even should the interrupt service routine become blocked. It is usually possible to prevent re-entrance to the routine prior to its completion. Performance of reactive systems is often measured according to *latency* (the delay between interrupt signal and commencement of service routine) rather than data or instruction throughput.

In his seminal book, Hoare discussed binary operators that capture both interruption and *alternation*, whereby some external signal causes behaviour of a system to switch between two processes [10, #5.4]. Upon change-over, the state of the interrupted process is preserved, allowing later resumption (interrupting its partner). This differs from the behaviour described earlier. First, the signal lies outside the alphabet of either process and must be repeated to cause resumption. Second, there is no prioritisation. Third, it is not immediately clear what

happens should either component process terminate. An interrupt service routine sometimes *disables* further interruption. It may even disable interruption by other events.

A new operator has been proposed by the author, along with a corresponding programming construct, that is intended to reflect the behaviour of non-re-entrant prioritised vectored interruption, and better serve the abstraction of reactive systems [15]. A companion paper [11] seeks to clarify the semantics of such prioritised alternation (termed pri-alternation, for convenience). It also provides a complete description of operator and construct.

We are already well-equipped to abstract the behaviour of processes composed under pri-alternation, which can be seen as complementary to parallel composition. (The designer must choose whether to provide two services (or bunches) via two basic processes arranged in parallel or via the same processes in pri-alternation.) Communication with an interrupt service routine is easily abstracted as the conduct of a single service. The interrupt signal may be considered a request for that service – the first in its characteristic sequence. Response is captured by the remainder of that sequence. Finally, the behaviour of the interrupt service routine may be captured via a process that is typically cyclic about interruption and completion. The question of termination is addressed in the companion paper.

3.2 Cyclic Dependency

Pri-alternation may be easily added to a service digraph by the use of a second kind of directed arc, representing the possibility of interruption of one process (node) by another. A *prioritised service digraph* (PSD) can thus be drawn using one colour for each kind of arc — here, gray for interruption and black for service.

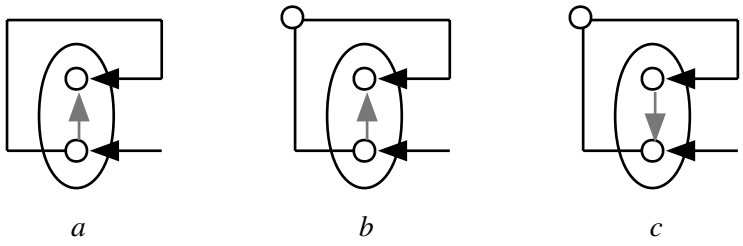

Figure 2: Three circuits within a prioritised service digraph (PSD).

With regard to deadlock in systems with PSA, consider the three circuits shown in Figure 2. Each diagram (*a*, *b*, and *c*) depicts two processes composed under prioritised alternation. In *a*, for example, a request for the service provided by the upper component will cause interruption of the lower one, as indicated by the gray arrow between them. The lower component will resume only when provision of that service has been completed.

Circuits of type *a* will clearly deadlock. Service cannot progress between two processes that alternate. If they alternate, they cannot be concurrent, and thus cannot synchronously communicate. Any attempt to connect services in this way will be denied by condition S4, which forbids loops, and by the SADR, which forbids circuits of any length.

Circuits of type *b* are permitted by the SADR and are inherently safe. Communication between upper and lower components of the alternation is mediated by the external process shown. It is free to gain a complete service from the upper component before proceeding to complete a service to the lower one. However, success depends upon the orientation of the gray arrow.

Those of type *c*, where the gray arrow is reversed, are still permitted by the SADR but are clearly unsafe. The upper component will *not* be at liberty to provide service to the external process. It must await completion by the lower one before it may resume. That will never occur.

To deny such eventualities, a new design rule is required for systems with *prioritised* service architecture.

Design Rule 2 (Prioritised Service Architecture Design Rule 1). *(*PSA DR1*)*
No directed circuit of dependent services and interruptions is permitted.

Note that this subsumes the earlier SADR, as a PSD subsumes a SD. PSA DR1 is thus the only design rule required, and the PSD is the only system design description required.

Prioritisation increases the scope for deadlock. As yet, deadlock-freedom cannot be guaranteed. PSA DR1 is a necessary but not a sufficient condition.

3.3 Priority Conflict

One other aim remains to be met — that prioritisation in one process should not conflict with that of another. Prioritisation here applies to services, not individual communications or processes, and forms part of the interface between processes. To be more precise, it is applied to server bunches, which are enumerated accordingly. The priority of each client connection is determined by the server bunch which, directly or indirectly, depends upon it.

Definition 6 (Priority Conflict). A *priority conflict* exists when the provision of some service is dependent upon the consumption of another of lower priority.

Inversion is one possible consequence, where an intended prioritisation is reversed as a result of parallel composition.

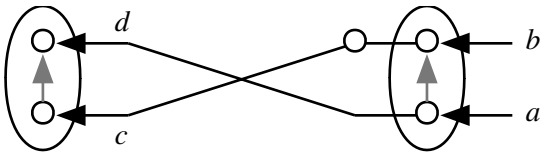

Figure 3: Priority inversion and potential deadlock in a PSD obeying PSA DR1.

Figure 3 shows how deadlock can arise from priority conflict even when PSA DR1 applies. Suppose a request for service *b* is made. The lower component of the right-hand alternation will be interrupted and service commenced. If it is dependent upon the service *c* then a request for that will be made, via the intervening node. All will be well unless the service *a* was already in progress and dependent upon *d*. If a request for *d* was made first at the left-hand alternation, deadlock will ensue.

Figure 3 also illustrates the futility of merely enforcing a local interface. No pair of processes can be found in direct conflict. More than a local connection rule is required.

The PSD has two dimensions – one representing interruption (gray), and one service (black). It will prove useful to require a normal form where all arcs are drawn oriented in common according to colour. For example, all gray arcs might be aligned vertically, with interruption proceeding upwards, and all black arcs aligned horizontally, with provision, say, on the left. Figure 3 is depicted in this fashion.

Design Rule 3 (Prioritised Service Architecture Design Rule 2). *(*PSA DR2*)*
When all arcs of the PSD *are aligned according to colour, no pair of dependencies may cross.*

This is clearly equivalent to a requirement that any prioritization of services, established via dependency, is consistent with that of the servers of any process to which they are connected. In other words, it must be possible to draw the PSD without any arcs of a given colour crossing. Freedom from priority conflict, and thus inversion, follows directly. (The proof is trivial but included for completeness.)

Theorem 3 (Prioritised Service Network Theorem 1).
Every system with PSA *that obeys* PSA DR1 *and* DR2 *is free of priority conflict.*

Proof. In the absence of a directed circuit in the SD (PSA DR1), it is possible to follow any dependency to its conclusion. Arcs along any dependency may be enumerated by back-tracking and decrementing the label each time an interruption is traversed in the corresponding PSD. The algorithm actually employed must take account of the fact that each arc may lie along multiple dependencies. For example, an existing label can be adopted and a correction applied back up the chain. It is thus possible to enumerate the global priority of every service.

It is therefore also possible to verify that the interface of each component is satisfied and thus that every system is free of priority conflict. □

PSA DR2 requires that the PSD can be *drawn* on a plane. One dimension corresponds to prioritisation (interruptability), the other to service dependency. It is proposed that this will also guarantee deadlock-freedom.

Theorem 4 (Service Network Theorem).
Every system with PSA *that obeys* PSA DR1 *and* DR2 *is free of deadlock.*

No proof is offered here of this theorem. This is because it would depend upon a formal definition of prioritised alternation — something which cannot be given using the failures model of CSP alone. (The formal definition of prioritised alternation is addressed in the companion paper [11].)

Formal conditions now govern whether any proposed system extension is admissible. It must be possible to incorporate the new subsystem within the existing (2-colour) PSD without dependency circuit or crossover. A suitable tool could verify this, still without the need to deploy mathematical skills in its application. The significance can be appreciated when it is remembered that around 98% of programmer activity is devoted to extending existing systems rather than constructing new ones.

As was noted earlier, performance of reactive systems frequently highlights latency rather than throughput. While physical time has not been introduced to the abstraction model presented here, in practice it remains possible to verify any latency requirement. A compiler will have knowledge of all aspects of the implementation of each pri-alternation. This can easily be extended to the timing parameters of the platform concerned. Thus verification of latency requirements could easily be accommodated within an appropriate programming environment.

4 Conclusion

Prioritised service architecture (PSA) is proposed for the abstraction of concurrent systems with a wide range of application, including those which are reactive (event-driven). It is suitable for the capture of both specification and design, where decomposition is safeguarded against deadlock and priority conflict by formal design rules. Direct implementation will be possible via the Honeysuckle programming language.

Figure 4 depicts a system with an interface comprising three components, each comprising a three-server bunch and a single dependent client. A prioritisation is also indicated. Provision of any service in bunch one will be pre-empted by a request for any in bunch two or three, for example.

PSA inherits the benefits of service architecture (SA). Not only are SA systems inherently deadlock-free, but they also are *compositional*; every component is a valid system, and every

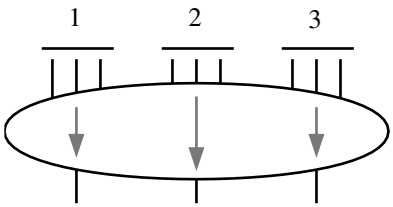

Figure 4: A prioritised service interface (PSI).

system a valid component. Components may be interchanged according only to their service interface.

Alternation provides a secure means by which to provide feedback within a system, avoiding the limitation to "process tree" structure inherent with client-server architecture defined according to data-flow. It also provides a simple, and natural, way to incorporate and express prioritisation.

Hybrid systems, where some components guarantee deadlock-freedom by enforcing other design rules, such as *cyclic order protocol* [4], can be achieved by exploiting the *Network Composition Theorem* of Brookes and Roscoe [5].

In addition to completing the definition of, and implementing, Honeysuckle, further work continues to address additional guarantees regarding pathological behaviour, and the issue of interference.

Acknowledgements

I am very grateful for many valuable and stimulating discussions with Jeremy Martin and Sabah Jassim.

References

[1] Jeremy M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, Hunter Street, Buckingham, MK18 1EG, UK, 1996.

[2] A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. Technical Report PRG-57, Oxford University Computing Laboratory, 8-11, Keble Road, Oxford OX1 3QD, England, 1986.

[3] Peter H. Welch, G. Justo, and Colin Willcock. High-level paradigms for deadlock-free high performance systems. In R. Grebe et al., editor, *Transputer Applications and Systems '93*, pages 981–1004. IOS Press, 1993.

[4] Jeremy Martin, Ian East, and Sabah Jassim. Design rules for deadlock freedom. *Transputer Communications*, 2(3):121–133, 1994.

[5] S. D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. *Distributed Computing*, 4:209–230, 1991.

[6] C. B. Jones. Wanted: A compositional model for concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, Monographs in Computer Science, pages 1–15. Springer-Verlag, 2003.

[7] Ian R. East. The Honeysuckle programming language: An overview. *IEE Software*, 150(2):95–107, 2003.

[8] Per Brinch Hansen. *Operating System Principles*. Automatic Computation. Prentice Hall, 1973.

[9] Jeremy M. R. Martin and Peter H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(3):1–18, 1997.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall International, 1985.

[11] Ian R. East. Towards a semantics for prioritised alternation. In East and Martin et al., editors, *Communicating Process Architectures 2004*, Series in Concurrent Systems Engineering. IOS Press, 2004.

[12] A. W. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice-Hall, 1998.

[13] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley, second edition, 2002.

[14] Gottlob Frege. Über sinn und bedeuting (on sense and reference). *Zeitschrift fr Philosophie und PhilosophischeKritik*, 100:25–50, 1892.

[15] Ian R. East. Programming prioritized alternation. In H. R. Arabnia, editor, *Parallel and Distributed Processing: Techniques and Applications 2002*, pages 531–537, Las Vegas, Nevada, USA, 2002. CSREA Press.