Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Streams and Signals

Dave Otway

Abstract

The endpoints of high bandwidth synchronised data streams used to be exclusively hardware devices, but increased processing power now makes it possible for them to be implemented in software. The increasing integration of multi-media interfaces into mainstream applications requires the integration of such stream endpoint software into general purpose computing systems.

Real-time systems have traditionally been implemented as specialised stand-alone systems. Increasing integration of business functions requires that they will have to interact with and be managed by non real-time systems.

The management of switching systems is the biggest distributed application.

All of the above require an architecture where synchronous parallel systems of limited scale can interact with and be managed by a very large scale distributed asynchronous system.

This document discusses some possible extensions to the ANSA Computational Model based on research into reactive systems.

APM.1393.02 **Approved** 12th January 1995

Architecture Report

Distribution:
Supersedes:
Superseded by:

Streams and Signals

Streams and Signals

Dave Otway

APM.1393.02

12th January 1995

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

1.1 Purpose

This document discusses some possible extensions to the ANSA Computational Model (ACM) [APM.1001]. The objectives of these extensions are the programming of stream endpoints and synchronous systems. These objectives have much in common and both need to be integrated into the asynchrony and concurrency of distributed systems. Considering both objectives together, increases the possibility of finding common solutions.

The *focus* of this document is on the semantics of computations, not on how they might be engineered or on what technology is required to support them.

1.2 Audience

This document is aimed at a specialised technical audience: in particular, researchers, architects and designers of execution, compiling, construction and verification tools concerned with the sending, receiving and control of externally synchronised data in a (generally asynchronous) distributed system; and in general, anyone concerned with the *semantics* of distributed computations.

1.3 Relevance

The endpoints of high bandwidth synchronised data streams used to be exclusively hardware devices, but increased processing power now makes it possible for them to be implemented in software. The increasing integration of multi-media interfaces into mainstream applications requires the integration of such stream endpoint software into general purpose computing systems.

Real-time systems have traditionally been implemented as specialised stand-alone systems. Increasing integration of business functions requires that they will have to interact with and be managed by non real-time systems.

The management of switching systems is the biggest distributed application.

All of the above require an architecture where synchronous parallel systems of limited scale can interact with and be managed by a very large scale distributed asynchronous system.

1.4 Background

1.4.1 Previous ANSA work

The work done in the ISA project on integrating multimedia into the ANSA architecture is reported in [APM.TR.28]. This describes some experimental

changes made to ANSAware to accommodate multimedia streams. It concentrates on the engineering and Quality of Service issues. The programmer interface is presented as extensions to PREPC.

1.4.2 ODP reference model

Part 3 of the Basic Reference Model of Open Distributed Processing [ODP-3] includes the concepts of signals and stream interfaces. This draft standard has similar origins and objectives as the ANSA work described in this document.

1.4.3 Research into reactive systems and synchronous languages

The synchronous approach to reactive systems is explained in [INRIA] and the Esterel synchronous programming language introduced in [BOUSSINOT Sep91].

1.4.4 CNET

Work done at CNET in applying synchronous languages to the problem of adding streams to the ANSA architecture was reported in [CNET Feb93].

This was then integrated with Quality of Service guarantees and binding in [CNET June93].

1.5 Motivation

Large distributed systems are unavoidably heterogeneous, federated, asynchronous, concurrent and above all non-deterministic. A computational model for such a system must concentrate on abstracting over most of the underlying complexity.

Real-time systems are inherently complex and require predictable access to shared resources.

The danger is that a computational model which caters for a mixture of both kinds of system would be too complex to use for large non real-time systems and too unstable to use for real-time systems. In particular, it is not feasible to rely on testing to guarantee the integrity of such a mixed system, because it is impossible to reliably reproduce any given behaviour.

In this context the reactive model [§3.1] of synchronous systems used by such languages as Esterel [BOUSSINOT Sep91] is attractive because it produces deterministic parallel programs whose behaviours are reproducible and whose execution times are predictable. Both these properties are essential if real-time systems are to be successfully integrated with large scale distributed systems.

In addition, the formal semantics of Esterel are designed to reject programs which are non-deterministic or have no solution, and to always generate an efficient implementation that avoids the use of hypothesis and backtracking.

Reactive systems, therefore, have the right properties to cope with the enormous complexity of large scale distributed systems. The greatest problems will occur, as always, at the boundaries between the synchronous and asynchronous parts of a system.

The obvious boundary to consider is that between the interaction and construction (sub)models of the ACM. This would result in a common interaction (sub)model and two alternative construction (sub)models;

synchronous and asynchronous. Although this is attractive in its simplicity, it doesn't deal with streams.

In particular, the problem of interfacing a synchronous stream to an asynchronous object requires solutions which have much in common with the reactive model.

Accordingly, this document explores the extensions required to the current ACM to enable it to model objects with both synchronous and asynchronous components.

It should be noted that Esterel has taken a limited step in the same direction with its exec primitive for executing asynchronous tasks, and that Reactive C [BOUSSINOT April91] comprehensively combines asynchronous and synchronous programming by adding reactive constructs to C.

1.6 Scope

This document only deals with computational model semantics. It assumes, based on the Esterel and Reactive C experience, that efficient implementations can be made available.

Only software stream endpoints are considered. The binding of stream endpoints implemented in hardware is not considered. The binding process considered is only that of the local endpoint; the construction of switches is a different topic and is not discussed here.

Synchronous programs will need specialised engineering and technology, but this document makes no attempt to define them.

Synchronous programs will also require specific guarantees from their environment [§1.7] and will be limited in their portability and configuration; but this document does not address these issues.

Since this document deals only with the semantics of the computational model, it uses the language of the computational model DPL [APM.1014 & APM.1015] to program the examples. This does not imply that the proposed additional constructs are only applicable to DPL. The most likely implementation of these concepts is as a preprocessor to a language such as C++.

This document is exploratory and does not attempt any formal definitions.

1.7 Related work

The work on extending the computational model is being done in conjunction with the following work.

1.7.1 Explicit binding

An explicit binding model is being developed for binding non-operational interfaces and for specifying Quality of Service guarantees on all interfaces. This will be reported in [APM.1239].

1.7.2 Quality of service

The QoS model being developed for real-time is described in [APM.1151].

1.7.3 Real-time engineering

The prototyping work on real-time engineering extensions to ANSAware is reported in [APM.1222].

2 Streams

This chapter examines the requirements for adding streams to the computational model, without considering the issue of synchronisation.

The computational model is only concerned with interfacing programs to stream endpoints. Stream binding, monitoring and control is handled by the engineering [APM.1239] and invoked via interfaces which conform to the computational model. Independently of whether their endpoints interface to programs, stream bindings may be managed by programs conforming to the computational model.

2.1 Flows

Streams consist of unidirectional data flows. Each flow has at least one source endpoint and one sink endpoint. Each endpoint may or may not interface to an object which conforms to the (enhanced) computational model.

Each data flow may be continuous or intermittent. Continues data flows have a Continues Bit Rate (CBR) and are synchronous. Intermittent data flows have a Variable Bit Rate (VBR) and are asynchronous.

An asynchronous data flow may be engineered over a higher average bandwidth synchronous data flow by padding. A synchronous data flow may be engineered over higher average bandwidth asynchronous data flow by buffering.

All data flows between non physically co-located objects have a transmission delay. Unavoidable imperfections in the implementation cause even synchronous data flows to exhibit some variation in their transmission delay, known as jitter.

2.2 Bindings

The source(s) of a data flow must be bound to its sink(s) before data can pass along it.

The implicit binding semantics for operational (as opposed to stream) interfaces in the existing ACM were designed to have good scaling characteristics and to optimise the provision of a single Quality of Service (QoS). A binding between a client and a server is not completed until the client first invokes an operation on the server. Most of the client's and all of the server's communication resources are reclaimed when the binding becomes dormant.

These implicit binding semantics enable applications that store millions of bindings on behalf of millions of clients to be constructed without tying up vast amounts of communication resources.

But stream bindings have very different characteristics: they are not expected to scale to more than a modest number of endpoints (without requiring an underlying broadcast technology), and they will require a wide variety of Qualities of Services.

Applications using streams may sometimes need to establish confidence in the QoS of the binding(s) in advance of communication taking place. They may require all of a group of bindings or none of them. They may wish to try an alternative binding, coordinate the establishment of a set of bindings or simply avoid the initial jitter that binding on first use would cause.

Because the application programmers need control over the timing of stream bindings, they must be explicit and each endpoint must establish its own end of the binding [APM.1239].

2.3 Direction of flow

With operational interfaces, the bindings are asymmetric, in that the server creates a service instance and an (uncompleted) binding to it, which is distributed to potential clients. Clients may then initiate an interaction (which completes the binding) and the server responds.

But with unidirectional flows, there is no obvious correlation between the application offering a service and the pattern of flow. For instance, the service offered by a video camera may be a flow that is always on, or one that must be explicitly switched on and off via a separate control interface.

Therefore, in a stream interface the server offering the service may be either the source or the sink of a flow and the client must take the opposing role.

2.4 Multiple flows

Should multiple data flows be allowed in the same stream?

To provided simple to use binding mechanisms for related flows, it is necessary either to provide some kind of interface grouping concept which enables a set of bindings to be passed around and manipulated as a single entity, or to allow multiple data flows to be included in a single stream.

The different flows will need to be distinguished at bind time so that different Qualities of Service may be applied to each flow; but when sending and receiving frames, the mechanism that is required to differentiate between multiple frame formats in the same flow should also be able to cope with multiple flows, as long as there are no naming clashes.

There are thus distinct binding convenience benefits in allowing multiple flows and only a minor naming and grouping problems to be solved.

2.5 Bidirectional streams

Should data be allowed to flow in both directions in a single stream?

This is very similar to the multiple flows issue. The applications could construct two back to back streams, but would find it much more convenient to refer to and bind them as a single entity. A lot of the underlying

communications technology is also duplex, so forcing a division in the engineering would be cumbersome.

2.6 Frames

Asynchronous flows which are interfaced to application programs have application defined frames, which are the units of data produced or consumed by the application in a single processing step.

Some synchronous flows which are interfaced to application programs may also have natural frames which provide a suitable application processing unit (e.g. a video frame); others may not (e.g. speech).

Processing a continuous bitstream bit by bit is unlikely to be the most efficient way of extracting information from or generating a flow. So even when there is no natural framing, the application designer must take a decision about the most appropriate size of data unit to process in each processing step.

The definition of a frame, in particular its start and end, will vary considerably, sometimes dynamically. Some frames will be defined by the technology of their transmission, generation or consumption. Others may well be defined by the communications engineering using such mechanisms as time, size, format or delimiters. Those flows which have no natural framing may leave the definition of a frame to the application program.

From the computational point of view it is simplest (and most efficient) to require the framing to always be done by the underlying engineering or technology. In those cases where the application is able to define the framing, this can be specified declaratively, and its implementation kept firmly in the communications engineering.

Thus all flows can be regarded as a sequence of frames where they interact with an application program. Indeed, divorcing the applications from the mechanics of framing makes them simpler and more portable, as well as allowing the framing to be done more efficiently.

Note that this discussion does not imply anything about the framing characteristics of streams whose endpoints do not interface to application programs.

2.7 Frame formats

Should all frames in a data flow be the same format?

A video flow consists of frames which in the main differ only slightly from the previous one, but sometimes there is a complete change of scene. Such a flow could be more efficiently transmitted if most frames were sent as a delta on the previous one unless a complete frame was smaller.

Compression techniques vary in their efficiency depending on the characteristics of the data. A flow with multiple frame formats would enable each frame to be compressed with the most efficient algorithm.

Multiple frame formats would enable application level multiplexing to be used, where a number of flows are mapped into a single flow. They would also enable in band signalling to be implemented by inserting control frames into the flow of data frames.

Given that a computational model that allows multiple frame formats can easily cope with technology which only provides a single format, while the reverse situation forces the application programmer to dive deep into the engineering to split the flow; it would be prudent to allow multiple frame formats in a flow.

Given this, the extended computational model must provide some way of differentiating between frame formats at both source and sink endpoints. This can be done by numbering or naming the distinct frame formats. Numbering would preclude the use of conformance based type checking [§2.12], so naming is preferred.

2.8 Frame arguments

The data in each frame must also be described. Arguments are the natural way to do this. What form should frame arguments take?

In the existing ACM, arguments in operations are bindings to other interfaces. Frame arguments have the same requirements as operation arguments, so they should also be bindings to (operational or stream) interfaces.

Should multiple arguments of different types be allowed? Why not?

2.9 Sources

If the client of a stream is the source of one of its flows then interfacing to the flow requires only a small relaxation of the operational invocation semantics of the ACM. The stream binding can be invoked with the frame name and arguments to transmit a frame down the flow. The only change to the operational invocation semantics is that the calling activity will not block awaiting a termination.

If the server of a stream is the source of one of its flows then the ACM semantics provide no way of identifying the server's binding to a stream. [Operation terminations are returned within the context of an enclosing operation invocation; so their return binding is implicit.]

But..., streams are not implicitly bound; both endpoints have to explicitly create their end of each new binding to a stream (instance). Therefore, for stream interfaces, the server will always have a local binding to the stream that it can invoke transmissions on.

2.10 Sinks

If the client of a stream is the sink of one of its flows then interfacing to the incoming flow has no parallels in the ACM. There is no despatching mechanism for clients and terminations are delivered via a blocked invocation.

The client always has a binding to the stream and can interface to the incoming flow via some form of read function[1] [§3.11].

1. This is the only place where a discussion of streams can't be separated from signals; please accept this vague handwave until all is revealed in the signals chapter.

If the server of a stream is the sink of one of its flows then interfacing to the incoming flow could be done using the operational dispatching semantics of the ACM.

But this would have the side effect of increasing the concurrency in the system because the caller does not block as it does for an operational interface. A more serious problem is that the dispatching mechanism provides no way of synchronising with different flows and time signals. Also, for reasons of symmetry it would be better to provide read functions for sinks in servers as well as clients.

2.11 Stream signatures

Should streams be typed?

The argument for type checking operational interfaces is just as valid for stream interfaces; namely the discovery of errors at compile (or bind) time rather than run-time.

Should stream type checking be equality or conformance based?

Again the same arguments apply to streams interfaces as to operational interfaces; type checking for equality is too restrictive and results in redundant re-building. It is almost impossible to upgrade a large scale permanently available system without subtyping.

If stream type checking is to be conformance based then the frame differentiation cannot be done by numbering, because the source cannot know how many frame formats its potential sinks may have. The frame formats must therefore be named.

The frame arguments must also be typed; but they do not need to be named because frames are never dispatched.

Combining the above gives a frame signature defined[1] along the lines of:

```
frame          = frameName "(" { typeExpression } ")"
```

and a flow signature is then simply a list of frame signatures and a direction indicator:

```
flow           = direction "(" { frame } ")"
direction      = ">>" | "<<"
```

where ">>" indicates an outgoing flow and "<<" indicates an incoming flow.

Flows need to be combined into streams and a stream signature would have a structure like:

```
stream         = "stream" "(" { flow } ")"
```

Since frames are never dispatched, they need no bodies and this form of signature will therefore serve for both a client or a server, although the direction of each flow would be reversed. The client/server distinction is much weaker for streams than for operational interfaces and only means which application publishes its stream interface. Whichever way round this

1. in BNF with the extension that {} enclose expressions that may occur any number of times, including zero

relationship is, both stream endpoints must be explicitly bound before communication can take place.

2.12 Type checking

Adapting the conformance rules for operational interfaces to flows requires that for a sink to conform to a source:

- the sink has at least the frames that the source has
- each frame in the source has the same number of arguments as the frame with the same name in the sink
- each argument in each frame in the source conforms to the corresponding argument in the corresponding frame in the sink

In terms of the type system, frames are operations without any terminations.

2.13 QoS

Quality of Service specifications can be applied to the stream as a whole by providing a QoS argument to the explicit binding operation [APM.1239]; but it is more convenient to apply QoS specifications to the various components of a stream by including attributes in the signature:

```
frameSignature     = frameName [attributeList]
                                    "(" { typeExpression } ")"
flowSignature      = direction [attributeList]
                                    "(" { frameSignature } ")"
streamSignature    = [attributeList] "(" { flowSignature } ")"
```

Different QoS can be provided for each distinct binding by parameterising the attributes.

The detailed QoS specifications are discussed in [APM.1151].

2.14 Mixed interfaces

Should flows and operations be included in the same interface?

This doesn't present any obvious computational difficulties; but it may be overcomplicating the engineering. It is really a trade-off between the extent of the engineering complications and the strength of the requirements. More experience is needed with both before reaching a conclusion.

2.15 Buffering

The frame concept separates application from engineering buffers. The engineering can split, concatenate or compress the application buffers before transmission

If the application produces or consumes data in small chunks then it may be desirable to provide a pipe interface which hides the framing. This can easily be done by library routines so should not be added to the computational model

3 Signals

This chapter examines the requirements for providing synchronisation in the computational model by the addition of signals, without considering the issue of streams.

3.1 Reactive model

A reactive system is defined as one that continuously interacts with its environment. When a reactive system receives input events, it responds by producing output events. A reactive system only produces output events in response to input events. Unless it receives input events, it does nothing. A reactive system can be viewed as a “black box”.

The execution of a reactive system is divided into instants which are the discrete moments when it reacts to its input events. A reactive system experiences a sequence of these instants as it is being executed. At each instant, a reactive program will consume all the input events that arrived since the previous instant and produce all the output events that they give rise to.

3.2 Synchronous hypothesis

The basic reactive model can be constrained by the synchronous hypothesis, which states that all reactions are instantaneous. This requires that a set of output events be synchronous with the set of input events that gave rise to them, as if they were produced by an infinitely fast machine.

This theoretical model can be realised in practice by ensuring that reactions are atomic. Successive reactions can be prevented from interfering with each other by ensuring that they cannot overlap; so that there is no possibility of activating a system while it is still reacting to an earlier activation.

The synchronous hypothesis simplifies reasoning about and programming reactive systems by removing all concurrency between successive reactions; which otherwise would be a source of non determinism.

3.3 Instantaneous communication

When reacting to a set of simultaneous input events, it would be simpler to program the reaction using parallel threads rather than to construct a state machine for all possible combinations.

Some of these threads will not communicate (interfere) with each other. Their execution can be serialised in any order, or they can safely be executed in parallel with each other.

Other threads will need to communicate with each other. This inter-thread communication can be modelled as instantaneous internal events, which communicate between threads (taking zero time) in the same instant that the external input events are consumed and the external output events are produced.

If there are no causality problems [BOUSSINOT Sep91] in the events used by a group of inter-communicating threads, then the execution of the threads can still be serialised even through they must be interleaved.

3.4 Determinism

The benefit of the synchronous hypothesis is that it enables the construction of programs whose behaviour is deterministic (and therefore reproducible). By deterministic we mean that the program will produce the required set of output events in response to each possible combination of input events and that the resources to do this (including processor time) will be bounded and can be calculated in advance for any given execution environment.

As long as these resources are guaranteed to be available, a synchronous program can be deterministic even in an asynchronous system.

3.5 Signals

Each input or output event in a reactive system requires communication with its environment. Such communications need to be distinguished from one another and to receive or transmit data. Each communication must also be a discrete message so that the system can determine when to react to or emit it.

This form of communication can be most generically modelled by named signals with typed arguments.

But programs need to control exactly when and to which combinations of signals they will react. The combining of signals can be done by defining a reactive event to be a specified set of signals; and the timing can be controlled by representing clocks as signals, which can then be combined with other signals.

3.6 Communications paradigm

External signals between a synchronous program and its asynchronous environment have to be buffered and clearly identified as outputs or inputs.

The synchronous hypothesis requires internal communication within a synchronous program to be instantaneous. The Esterel language achieves this by broadcasting an internal signal to all its receivers.

However, this broadcasting cannot be extended to external signals in an asynchronous distributed system. Using a broadcast communication paradigm for internal signals would mean having different internal and external communications models, which would make it harder to change the configuration of a system by changing the assignment of objects to capsules and adjusting the granuality of externally visible instants.

The models need to be different in that any two-way instantaneous communications would always require co-location of both endpoints in the

same capsule; but a non-broadcast communication between clearly identified sources and sinks would allow much greater reconfiguration flexibility.

Broadcast communication requires global signal names, which also restrict reconfiguration flexibility by introducing the possibility of name clashes.

For the above reasons, it would be better (and more compatible with the current ACM) to group both external and internal signals into interfaces.

3.7 Specifying sources and sinks

Each signal has sources and sinks. For external signals, a program needs to specify which endpoint role it will play with regard to each signal. For internal signals, the program needs to identify which of its components will play which role for each signal. This could be done by specifying the direction of each signal in an interface, and requiring each party to a binding of this interface to identify itself as a transmitter or receiver.

Allowing multiple instances of signal interfaces would enable multiple bindings to similar services in the same object without signal name clashes.

3.8 Signal interfaces

A signal signature needs to specify the name, argument types and direction of the signal. A set of these signals signatures can then be combined into a signal interface signature.

```
direction      = “>>” | “<<“
Signal         = direction SignalName “(“ { typeExpression } “)”
SignalInterface= “signalInterface” “(“ { Signal } “)”
```

The direction “>>” indicates a signal which can be transmitted and “<<” indicates a signal which can be received.

3.9 Binding signal interfaces

Signal interfaces have no operation bodies so the only difference between the two ends of a binding is the direction of the signals in the interface. After the bindings have been established they will be in a peer-to-peer relationship with no restrictions, other than the signal directions, on which can initiate communication.

However, before a signal interface binding can be established, both parties have to create their own endpoint, exchange endpoint references and then create a binding between their own endpoint and the other party’s endpoint [APM.1239].

A simplified form of these bind operations (with no QoS statements) for one endpoint is:

```
    myend = signalInterface of ( >> transfer(String) )
; yourend = swap.endpoints(myend)
; s = myend.bind(yourend)
```

The endpoints can be swapped using any suitably defined operation or termination, and can be transferred via third parties.

3.10 Transmitting signals

Transmitting signals is straightforward and the operation invocation syntax could be re-used, but it would be clearer to differentiate transmissions from invocations:

```
transmission  = unit "!" signalName block
```

[The unit sub-expression resolves to the signal interface binding and the block provides the arguments.]

A transmission only blocks until the signal has been transmitted, or queued for transmission; depending on the local engineering. There is no reply.

3.11 Receiving signals

Receiving a signal is a completely new concept in the ACM, and there are a number of options for the blocking semantics of receiving, namely:

1. block until the signal is present and then read the arguments
2. interrupt the processing of a reaction if a specified signal is received, and optionally process that signal
3. block until the signal is present but don't read the arguments
4. test if the signal is present, but don't block or read the arguments
5. read the arguments if the signal is present, but don't block

Option 1 is clearly necessary and should cater for the majority of cases.

Option 2 is one of the basic reactive system concepts, the watchdog; which interrupts a reaction when relevant circumstances change.

Option 3 on single signals is not very useful (given the availability of option 1), but is very powerful when used with multiple signals and the availability of option 4.

The availability of option 4 and option 3 with multiple signals enables the specification of the start of a reaction and the processing of the signals that started it to be completely separated.

Options 5 doesn't add any capability to a combination of options 4 and 1.

So the minimum set of options is:

- a blocking read on a single signal
- a watchdog for a single signal
- a presence test on a single signal

and the useful extensions to be considered are:

- a non reading block on multiple signals
- a presence test on multiple signals
- a watchdog for multiple signals

Blocking on, testing for and aborting on multiple signals can be done for all or any of them. For instance, a wait for three signals may be defined as blocking only until any one of the three signals is received or until all three have been received.

Blocking on, testing for and aborting on any of a set of signals can be easily programmed using a concurrent expression list. Blocking or aborting for all of a set of signals and then collecting the arguments is much more difficult to program.

Providing a watchdog for any of a set of signals which processes only those signals which occur introduces significant complexity, but one which only fires if all signals occur is very simple.

Therefore the extensions will only be applied to all of a set of signals.

The semantics of receiving multiple signals are complicated by the fact that the signals may be received in different instants. For statements such as await [§3.13] and watchdog [§3.14] which do not act on received signals until the full set has been received, the signals are buffered as they are received and all of them are promoted to the instant in which the last one is received.

3.11.1 Reading single signals

A blocking read on a single signal could be done by an expression of the form:

```
reception      = unit "?" signalName
```

This blocks until the signal is present and then delivers the arguments. If the signal delivers arguments, a receiver would normally be the right hand side of an assignment or definition expression.

3.11.2 Testing combinations of signals

The basic test is for the presence or absence of a single signal. An expression which just refers to the signal, such as:

```
condition = "(" reception ")"
```

is true if the signal is present and false if it is absent. [The distinction between a blocking read and a presence test is provided by context; a reception in an expression list blocks and reads, a reception in a condition just tests.]

This single test can then be combined into a multiple test simply by:

```
condition = "(" { reception } ")"
```

which tests for the presence of all the signals. If any are absent then the condition is false. A signal may not be listed twice.

3.12 Synchronous expressions

In the asynchronous computational model, time moves forwards rather than backwards. The order in which a list of expressions is executed can be defined, but the time that the execution is started or the time it takes cannot be quantified.

In a synchronous system, time is divided into a series of logical instants. At each instant, a number of expressions are executed in response to the input signals present in that instant. These logical instants are ordered, but have no duration. They can be directly linked to "real" time by arranging for an external clock to generate signals which are input to the program.

A synchronous expression waits for a certain (combination of) signal(s) and is executed in the instant that the signal(s) are received. The simplest synchronous expression just waits for a signal to arrive and then receives it, such as:.

```
chime = clock?hour
```

This expression waits for the first time the hour signal is received then reads the argument and binds it to chime.

3.12.1 Sequential expressions

The following sequence waits for the first hour to be chimed, discards the argument and then rings a bell in the same logical instant:

```
clock?hour ; bell!ring()
```

3.12.2 Parallel expressions

Parallel reactions can be programmed using a concurrent expression list, for example:

```
[ clock?second ; clock?second ] || button?press
```

will block until a full second has elapsed and the button has been pressed, whatever the order of the signals.

3.12.3 Delimiting logical instants

There are a number of options for delimiting the boundary between the end of one logical instant and the start of a new one:

1. it is always delimited by an explicit stop statement
2. it is implicit in the structure of a control statement, [e.g. every iteration of a loop]
3. each blocking read or condition starts a new instant [but reading a signal that has been verified as present by a condition does not]

Option 1 only delays a reaction till the next instant, but this is an arbitrary length of time and as such is not nearly as useful as waiting for a specific signal. The real choice is between 2 and 3.

Option 2 requires that all reactions are scoped by specialised reactive constructs and therefore restricts the re-use of synchronous ACM constructs.

Option 3 is most natural and requires the least additional concepts, so it is the preferred one in the absence of any major difficulties.

3.13 Waiting for a combination of signals

Waiting for multiple signals can be done with an await expression controlled by a condition:

```
await     = “await” condition
```

This will block until all of the signals have been received. The signals may be received in any order and in different instants. The expression blocks until all

the specified signals have been received, but it also keeps the signals and makes them all present during the instant the block is released.

The outcome of an await expression is an anonymous termination with no results.

3.14 Watchdogs

Sometimes a sequence of reactions needs to be aborted because of changing circumstances. A watchdog expression will execute the watched block while watching for the condition to become true.

If the condition is already true at the start of the expression, the execution of the watched block will not be started. If the condition becomes true during the execution of the watched block, it will be aborted. In both cases, the alternative block will be executed.

```
watchdog = “during” block “watch” condition block
```

The execution of the watched block may be spread over many instants.

At the start of each instant the condition is evaluated and if false the watched block is allowed to execute for that instant. If the watched block completes its execution then the outcome of the watchdog expression is the outcome of the watched block.

If the condition becomes true at the start of any instant, the execution of the watched block is aborted (or not started) and the execution of the alternative block started. If the execution of the watched block is aborted (or not started) then the outcome of the watchdog expression is the outcome of the alternative block.

The condition semantics are the same as for an await expression in that all the watched signals are kept and made present in the instant the watched block is aborted.

A possible cause of non-determinism is if the signal(s) in the watched condition would also enable the completion of the watched block. The choice of semantics in this special case is:

1. the watched block is executed
2. the alternative block is executed
3. both blocks are executed
4. this situation is forbidden

Option 3 is unlikely to be useful, since the watchdog is designed to specify alternative behaviour, and it can easily be programmed by a following presence test [§3.15].

Option 2 is a more logical application of the simple semantics that the watched signal always aborts the watched block but is likely to be less useful than option 1, since if the watched block can explicitly be terminated by the signals(s) then this is clearly the required behaviour, otherwise why would the programmer have bothered. So option 1 looks the most useful and is the initial choice.

Option 4 is a bit draconian, but may be necessary if experience shows that option 1 causes too much confusion.

If watchdog expressions are nested (i.e. a watched block contains a watchdog expression), then the outer expression has the higher priority. If the outer watchdog's condition becomes true then the whole of its watched block, including whichever block of the inner watchdog expression is currently executing, is aborted. If both watchdog's conditions become true in the same instant then the outer watchdog's alternative block is executed and the inner one's isn't.

3.15 Presence test

The presence expression will test which signals are present during the course of a reaction without blocking.

```
presence = “present” condition
```

If the condition is true then the outcome is the named termination ->true(). If the condition is false then the outcome is the named termination ->false().

3.16 Loops

The sequence in [§3.12.1] would only ring the bell once. To ring the bell every hour requires a looping construct that iterates once for every instant that it is activated. The simplest form of loop is one that has no control on the number of iterations. So a loop to ring the bell every hour is [using the ACM minimalist approach of tail recursion]:

```
chimer = interface
          ( bell () ->()
               [ clock?hour ; bell!ring() ; chimer.bell() ]
          )
```

and a loop that goes ding every hour and dong two seconds later is:

```
chimer = interface
          ( ding_dong () ->()
               [ clock?hour  ; bell!ding() ; clock?second
               ; clock?second; bell!dong(); chimer.ding_dong()
               ]
          )
```

Note that this takes three separate instants to execute each iteration and that the instants are not necessarily consecutive ones. This is true even if the second signal is present in the same instant as the hour signal.

A loop that is controlled by a more complex condition can be constructed with the await expression.

```
v = interface
          ( refill () ->()
               [ await (hopper?full vat?empty)
               ; valve!open()
               ; v.refill()
               ]
          )
```

3.17 Locality

For a compiler (or other programming production tool) to be able to determine in advance whether or not a program will execute correctly with specified engineering, technology and resources under a given operating load, the programs execution paths must be bounded.

In a distributed system it is generally impossible to bound remote executions. So a simplistic view of synchronous programming would enforce the restriction that all operation invocations in synchronous expressions would be restricted to local interfaces running on the same engineering and technology, sharing the same resources and inspectable by the same tools.

To provide guarantees for distributed synchronous programs would entail detailed control over the construction, scheduling and communications with remote objects. This is the long term goal, but it will require an awful lot of pegs to be put in place before it is achievable.

3.18 Simultaneous internal signals

There is a problem with the use of thread serialisation to provide instantaneous internal communication [§3.3] when the same internal signal is transmitted by more than one thread in the same instant. The receiving thread has also been serialised and so can't cope with more than signal per instant without invalidating its execution bounds. It also can't cope with multiple sets of argument on a combined signal without sacrificing type safety.

This is not a problem with external signals because the thread serialisation used to provide instantaneous communication and thus bounded execution paths is confined to a capsule.

The problem with simultaneous internal signals is best avoided by getting the compiler to check for it as a programming error. Failing this a run-time error should be generated. The option of discarding all but one of the signals will destroy the determinism of the program, because there is no way of unambiguously specifying which signal an implementation should keep.

Note that the ability to receive simultaneous internal signals is not a requirement because the programmer can achieve the same effect by renaming all but one of the signals and explicitly receiving all of them.

3.19 Constructs considered but rejected

3.19.1 Traps

Esterel has a trap construct within which an exit expression will terminate the trap block. This is very similar to a loop with a break expression except that the exits are labelled rather than counting the level of nesting.

This construct has almost identical semantics to a watchdog where the watched signal is transmitted from within the watched block. The difference in Esterel is that such a signal may also originate from outside the watched block and because of Esterel's broadcast communication semantics, would abort the watched block.

The communication semantics proposed in this document are not broadcast. Both signal endpoints have to be explicitly bound to local identifiers which can be scoped so they are not available outside the watched block.

3.19.2 Disjunctive conditions

The proposed multi-signal condition is conjunctive because:

- it is hard to program out of single conditions,
- the complexity quickly explodes as more signals are added.

On the other hand, disjunctive conditions were not included because:

- they are easy to program
- multiple alternative blocks would have to be provided for a watchdog

4 Combining streams and signals

Streams and signals have a great deal in common and only a few minor differences. This chapter examines those differences and proposes solutions which enable stream endpoints to be controlled by synchronous programs.

4.1 Flows

Streams have a requirement to identify different component flows so that their direction can be specified and a particular Quality of Service requested for their binding.

Signals have no such requirement but would not be inconvenienced by having flows; just an extra level of brackets in their signatures.

Therefore a common signature would look like:

```
signal        = signalName [attributelist]
                                “(“ { typeExpression } “)”
direction     = “>>” | “<<“
flow          = direction [attributelist] “(“ { signal } “)”
stream        = “stream”  [attributelist] “(“ { flow } “)”
```

where frames have become signals and signal interfaces have become streams.

4.2 Persistence

Should signals persist until they are read or just for the instant in which they arrive?

Control programs are generally not interested in all occurrences of all signals, because signals are only of interest during specific circumstances and synchronous programs rely on having enough computing resources to be waiting for every signal that they are currently interested in. Therefore, signals should be discarded the instant after they arrive if nothing is waiting for them.

But streams are used for data transmission and it is not usually desirable that data is discarded just because the receiver is busy. This problem is usually overcome by buffering; but this is only a short term solution because if the data consistently arrives at a faster rate than it can be processed then any size buffer will eventually overflow.

On the other hand, the whole philosophy of supporting synchronous programs in an asynchronous system relies on being able to calculate the execution bounds in advance and to guarantee the provision of sufficient resources to processes all possible combinations of signals. Given this, a signal that arrives while the program is too busy to process it should be treated as a design error.

This strict attitude to overloads is desirable for synchronous streams, since the program could not recover from an overload anyway; and together with the control program requirement is enough to justify discarding unwanted signals as the default behaviour.

However, this would condemn programs processing asynchronous streams to provide resources to cope with the peak load all the time. Therefore signal buffering should be made available, but only when explicitly requested and sized via a QoS statement [APM.1151].

4.3 Summary of the proposed extensions

The full list of proposed extensions to the computational model are:

```
signal        = signalName [attributelist]
                           "(" { typeExpression } ")"
direction     = ">>" | "<<"
flow          = direction  [attributeList] "(" { signal } ")"
stream        = "stream" [attributeList] "(" {flow} ")"
transmission  = unit "!" signalName block
reception     = unit "?" signalName
condition     = "(" {reception} ")"
await         = "await" condition
watchdog      = "during" block "watch" condition block
presence      = "present" condition
```

The new kinds of expression, transmission, reception, await, watchdog, presence and stream would be added to the existing expressions:

```
expression    = activity | assignment | definition
              | handledBlock | initialisation | interface
              | literal | terminate | type | unit | transmission
              | reception | await | watchdog | presence | stream
```

and stream flows added to the type expression.

```
type          = "type" [attributeList] "(" [ { signature }
                                           | { flow } ] ")"
```

5 Examples

The examples in this chapter illustrate the use of the proposed stream and signal extensions to the ANSA Computational Model. To keep them simple and separate from the parallel work going on with explicit binding, all binding details have been omitted. Each example defines the stream types it needs and then provides an operation to process previously bound streams. The only exception is the internal timer stream in the mouse handler, which uses a binder operation which creates two local stream endpoints (one with the reverse type of the other), cross binds them and returns the two local stream bindings.

5.1 A simple mouse handler

A simple mouse has just one button and emits two signals, buttondown and buttonup. The mouse handler converts a pair of buttondown-buttonup mouse signals into a singleclick event signal if they are sufficiently close together. In the same way it converts a double pair of buttondown-buttonup mouse signals into a doubleclick event signal. Button mouse signals that are not close enough to form a pair are passed straight through as button event signals.

The handler converts a mouse stream into an event stream using a clock stream.

```
; mousetype = type ( << ( buttondown() buttonup() ) )
; eventtype = type ( >> ( buttondown() buttonup()
                          singleclick() doubleclick() ) )
; clocktype = type ( << ( tenth() ) )
```

The handler is constructed as an interface with a start operation whose arguments are bindings to instances of the three streams defined above. The operation first constructs a timer stream for internal communication within the handler.

```
; handler = interface
   ( start (mouse:mousetype clock:clocktype event:eventtype) ->()
       [ clickdelay = 5
       ; timerin timerout = bind.stream(type ( << ( done() ) ) )
```

A local interface is needed to provide the outer loop via tail recursion of the next operation. It also contains two other operations which just modulerise the implementation. The time operation counts tenths of a second clock signals and then terminates by writing the timerout!done event.

```
; h = interface
   ( time (tenths:Integer) ->()
        [ clock?tenth
        ; after  ( tenths.sign[1]() )
          handle ( zero() [ timerout!done() ]
                   positive(t:Integer) [ h.time(t) ]
                 )
        ]
```

The count operation is invoked after the initial mouse?buttondown signal in a sequence to count the rest of the signals in a sequence. If a complete doubleclick sequence is received then an event!doubleclick signal is transmitted and the operation terminates.

The counting block is aborted if a timerin?done signal is received before the sequence is complete. The incomplete sequence of mouse signals is then translated into the corresponding event signals. Note that when the last mouse signal received was a buttondown, the operation explicitly waits for the corresponding and inevitable buttonup.

```
count () ->()
   [ counter:Integer := -1
   ; during
        [ mouse?buttonup   ; counter := counter.add(1)
        ; mouse?buttondown ; counter := counter.add(1)
        ; mouse?buttonup   ; event!doubleclick()
        ]
     watch (timerin?done)
        [ after [ counter.sign() ]
          handle (negative(c:Integer)
                      [ event!buttondown()
                      ; mouse?buttonup
                      ; event!buttonup()
                      ]
                  zero()
                      [ event!singleclick() ]
                  positive(c:Integer)
                      [ event!singleclick()
                      ; event!buttondown()
                      ; mouse?buttonup
                      ; event!buttonup()
                      ]
                 )
        ]
   ]
```

The main loop explicitly waits for the mouse?buttondown signal that starts a possible single or double click. It then executes the timer and counter in parallel. When both have terminated it loops.

1. The sign operation on an Integer returns a zero(), positive(Integer) or negative(Integer) termination. For positive and negative terminations, the integer result is one closer to zero than the integer the operation was invoked on.

```
            next () ->()
               [ mouse?buttondown
               ; [ h.time(clickdelay) || h.count() ]
               ; h.next()
               ]
         )
```

The final expression kicks off the main loop.

```
      h.next()
   ]
)
```

5.2 An operation to signal converter

In order to demonstrate that the proposed changes to the computational model do not prevent synchronous programs from interacting with asynchronous ones, the following converter changes a stream based stack into an equivalent operational one. Given the existence of a stream stack server of the type:

```
streamStack = type ( >> ( push(Integer) pop() )
                     << ( pushed() popped(Integer) empty() )
                   )
```

The converter will create a equivalent operational service of the type:

```
operationalStack = type ( push(n:Integer) ->()
                          pop() ->(Integer) ->empty()
                        )
```

This is done by creating and returning an interface where each operation sends a signal equivalent to an operation request then waits for and returns the signals corresponding to the operation terminations.

```
convert = inteface
     ( stack(s:streamStack) ->(operationalStack)
          [ interface
               ( push(n:Integer) ->()
                    [ s!push(n) ; s?pushed ]
                 pop() ->(Integer) ->empty()
                    [ s!pop()
                    ; during [s?popped]
                      watch (s?empty) [->empty()]
                    ]
               )
     )
```

5.3 Merging streams

This example merges two simple incoming data streams into a single outgoing data stream and resynchs the output stream.

The data streams only have one signal which transmits a string of bytes. The timer stream just delivers a simple tick, which is conveniently of exactly the right frequency.

```
  DataIn  = type ( << ( data(String) ) )
; DataOut = type ( >> ( data(String) ) )
; Timer   = type ( << ( tick() ) )
```

The merge.twoway operation takes two input streams, an output stream and a timer as its arguments. It constructs a loop to merge the streams and then kicks it off.

The body of the loop waits for both input signals and the timer tick, if it gets all three then it concatenates the two input data arguments and transmits the combined argument on the output stream. If the timer tick signal is received before both input data signals then the watchdog checks if either of the two input signals have been received. If one of them is present then its argument is transmitted on the output stream; otherwise a null string is sent.

```
; merge = interface
   ( twoway (in1:DataIn in2:DataIn out:DataOut timer:Timer) ->()
       [ s = interface
          ( loop () ->()
               [ during
                    [ await (in1?data in2?data timer?tick)
                    ; out!data(in1?data.append(in2?data))
                    ]
                 watch (timer?tick)
                    [ after [ present (in1?data) ]
                      handle ( true()
                                   [ out!data(in1?data) ]
                               false()
                                   [ after  [ present (in2?data)]
                                     handle ( true()
                                              [out!data(in2?data)]
                                             false()
                                              [out!data(“”)]
                                            )
                                   ]
                             )
                    ]
               ; s.loop()
               ]
          )
       ; s.loop()
       ]
   )
```

References

[APM.1001]

Owen Rees: The ANSA Computational Model, APM Ltd., Cambridge (UK).

[APM.1014]

Andrew Watson *et al*.: DPL Programmers' Manual, APM Ltd., Cambridge (UK).

[APM.1015]

Dave Otway: DPL Reference Manual, APM Ltd., Cambridge (UK).

[APM.1151]

Guangxing Li: A Model of Real-Time QoS, APM Ltd., Cambridge (UK).

[APM.1222]

Guangxing Li: Some Engineering Aspects of Real-Time, APM Ltd., Cambridge (UK).

[APM.1239]

Dave Otway: The ANSA Binding Model, APM Ltd., Cambridge (UK).

[APM.TR.28]

C. Nicolaou: Integrating Multimedia into the ANSA Architecture,, APM Ltd., Cambridge (UK).

[BOUSSINOT April91]

Frederic Boussinot: Reactive C: An Extension of C to Program Reactive Systems, Software-Practice and Experience, Vol. 21(4), April 1991.

[BOUSSINOT Sep91]

Frederic Boussinot & Robert de Simone: The ESTEREL Language, Proc. of the IEEE, Vol. 79, No. 9, September 1991.

[CNET Feb93]

L. Hazard, F. Horn, J.B. Stefani: Some computational and engineering aspects of streams, ISA Project Report CNET/RC.W03.LHFH.001, February 1993.

[CNET June93]

L. Hazard, F. Horn, J.B. Stefani: Towards the integration of real time and QoS handling in ANSA architecture, ANSA Phase III Project Report CNET/ RC.ARCADE.01, June 1993.

[INRIA]

A. Benveniste, G. Berry: The Synchronous Approach to Reactive and Real-Time Systems, Rapports de Recherche No. 1445, INRIA, June 1991.

[ODP-3]

Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model, ISO/IEC JTC SC21 N8538.