Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE:	**Cambridge (0223) 323010**
INTERNATIONAL:	**+44 223 323010**
FAX:	**+44 223 359779**
E-MAIL:	**apm@ansa.co.uk**

ANSA Phase III

Streams and Signals (TC presentation june 94)

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 presentation proposes some possible extensions to the ANSA Computational Model based on research into reactive systems.

A write up of these proposals is contained in APM.1108

APM.1252.00.01 **Draft** 25 April 1995
Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Streams and Signals

Work in Progress Report

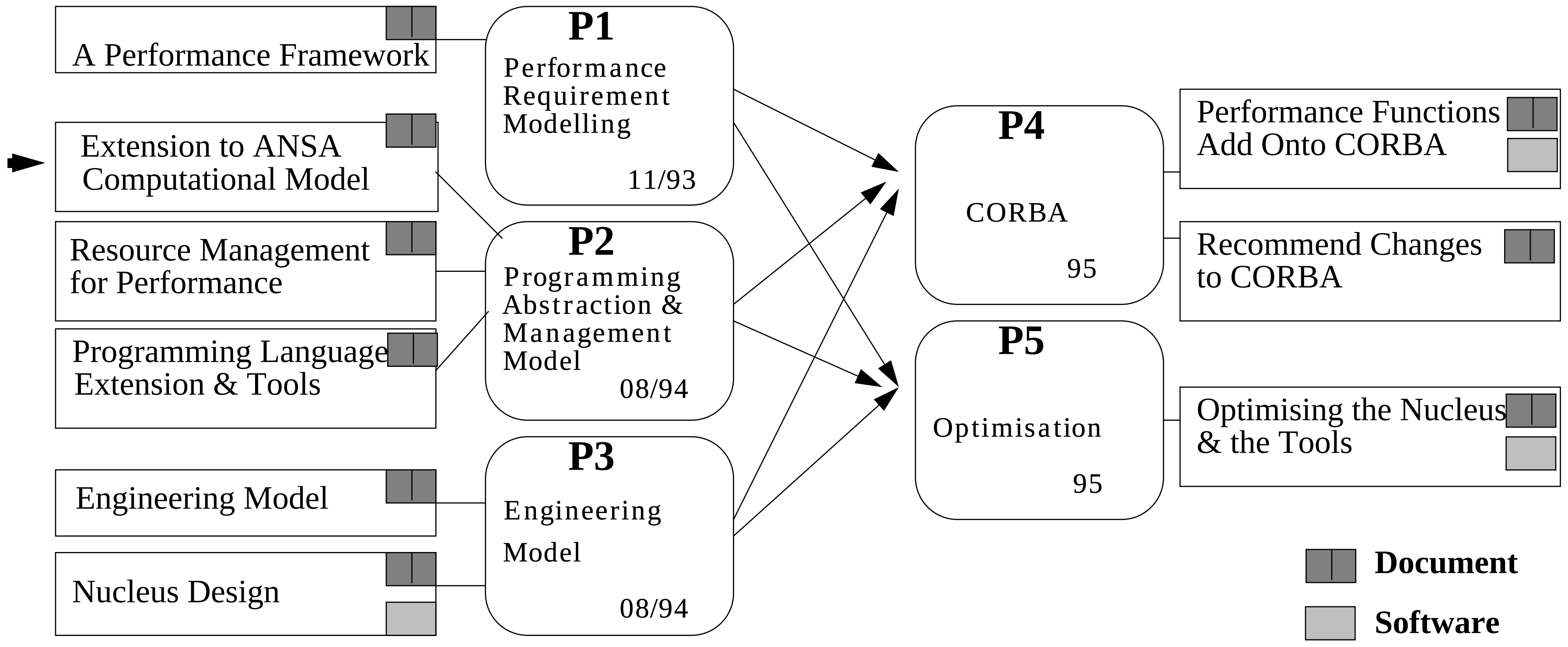
(Request for Comments APM:1108)

Dave Otway

djo@ansa.co.uk

Performance Group

Group Activities


A Performance Framework
Extension to ANSA Computational Model
Resource Management for Performance
Programming Language Extension & Tools
Engineering Model
Nucleus Design
P1 Performance Requirement Modelling 11/93
P2 Programming Abstraction & Management Model 08/94
P3 Engineering Model 08/94
P4 CORBA 95
P5 Optimisation 95
Performance Functions Add Onto CORBA
Recommend Changes to CORBA
Optimising the Nucleus & the Tools
Document
Software

Motivation

- endpoints of high bandwidth synchronised data streams used to be hardware
  - mips now available to process them in software
  - new media and formatting standards -> soft engineering
- multi-media increasingly used in mainstream applications
- integration of business functions requires real-time systems to interwork with non real-time systems
- synchronous programming research enables synchronous, predictable programs to be executed in a basically asynchronous system

Extending the Computational Model

- add new data concepts:
  - flows, frames, streams, signals
- add new synchronous programming concepts:
  - transmission, reception, condition, await, present, watchdog
- previous work:
  - IMACS, ODP Reference Model, synchronous programming (CNET, INRIA, Syseca)
- related work:
  - explicit binding, quality of service, synchronous engineering

Data Flows

- data flows are unidirectional from source endpoint(s) to sink endpoint(s)
- each endpoint may or may not interface to a software object
- the source(s) of a data flow must be bound to its sink(s) before communication can take place
- no correlation between sources / sinks and clients / servers
- combining multiple data flows into a bidirectional stream provides:
  - a simple binding reference for related flows
  - easy exploitation of duplex communications engineering and technology
  - abstraction of connection models

Framing

- asynchronous data flows have application defined frames
- some synchronous data flows may have natural frames, others may not
- even when there is no natural framing, the application designer must take a decision about the appropriate size of data unit for each processing step
- even when the framing is arbitrary it is simpler, more efficient and more portable to do the framing in the engineering [just like stubs]
- all data flows can be regarded as a sequence of frames at the point where they interact with an application program

[this does not imply anything about the framing characteristics of hardware endpoints]

Frame Formats

- multiple frame formats in the same flow enable:
  - frames which differ little from the previous one to be sent as a delta
  - the compression algorithm to be changed between frames
  - application level flow multiplexing / demultiplexing
  - in band control
- frames can be differentiated by names or numbers
  - numbers would preclude the use of multiplexing, in band control and conformance based type checking
- the data in a frame can be described by typed arguments
  - an argument is a binding to an (operational or stream) interface

Stream Signatures

- a stream signature that enables conformance based type checking is:

```
frame     =  frameName “(“ { typeExpression } “)”
direction =  “>>” | “<<“
flow      =  direction “(“ { frame } “)”
stream    =  “stream” “(“ { flow } “)”
```

- since frames are never dispatched
  - they don’t need bodies
  - their arguments don’t need names
- the same signature grammar can be used to describe both endpoints
  - with the direction of the flows reversed

Synchronous Programming

- a reactive system continuously interacts with its environment
  - its execution is divided into a sequence of discrete instants
  - at each instant it reacts to all its inputs by producing the outputs they give rise to
- the synchronous hypothesis model all reactions as instantaneous
  - simplifies reasoning by removing all concurrency between successive reactions
  - execution of communicating threads in the same reaction is serialised
- deterministic behaviour
  - bounded execution paths, calculable in advance
  - with guaranteed resources:
    - programs have predictable timing and reproducible behaviour
    - even in asynchronous systems

Signals

- each input or output must be a discrete message so the program can determine which instant it must react to it or emit it
- each input or output event is defined by a named signal with typed arguments
- each signal has a defined direction (in or out)
- communications are not broadcast, but grouped into signal interfaces
  - gives same local and remote semantics
  - enables bindings to be scoped
  - enables many instances of a signal interface to be used by the same program
- reactions are synchronised with real-time by providing input time signals

Signal signatures and binding

- **a signal interface signature consists of :**

```
direction =  “>>” | “<<“
signal     = direction signalName “(“ { typeExpression } “)”
signalInterface = “signalInterface” “(“ { signal } “)”
```

- **both parties create their own endpoint, exchange endpoint references and then establish an explicit binding from their own endpoint to the other party’s**

  - **A simplified binding establishment might be coded as:**

```
  mine = signalInterface( >> transfer (String) )
; yours = swap.endpoints(mine)
; s = bind.endpoint(type(>>transfer(String)),mine,yours)
```

Transmission and reception of signals

- transmitting signals is similar to invoking operations

```
transmission = unit “!” signalName block
```

  - which only blocks while the signal is queued - there is no reply

- receiving signals is done by an expression:

```
reception = unit “?” signalName
```

  - which blocks until the signal arrives and then returns the arguments

- sequential example:

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

- parallel example:

```
[clock?second ; clock?second] || [button?press ; bell!ring]
```

Waiting and testing for signals

- the basic expression for testing for the presence of a set of signals is:

```
condition = “(“ { unit “?” signalName } “)”
```

- a non blocking test to see if all of a set of signals are present is:

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

- a blocking wait until all of a set of signals have been received is:

```
await = “await” condition
```

  - the signals may be received in any order and in different instants
  - all signals are promoted to the instant in which the last signal is received

Watchdogs

- sometimes a sequence of reactions needs to be aborted
  - a watchdog expression will execute a block while watching for a condition to become true

    ```
    watchdog = “during” block “watch” condition block
    ```
- if the condition becomes true before the watched block terminates then its execution is aborted and the alternative block is executed
- if the watched block completes its execution then the watchdog expression terminates
- the condition semantics are the same as for await
  - all signals are promoted to the last instant
- if the (dynamically) last signal in the condition would also enable the completion of the watched block
  - the watched block is completed

Loops

- use tail recursion

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

Summary of proposed extensions

- **add the following new constructs**

```
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
```

- **extend expression and type constructs to accommodate the new constructs**

What next ?

- examine every possible interaction with existing computational model semantics
- formalise semantics of extended computational model
- apply synchronous extensions to TPP
- design run-time engineering
- prototype synchronous program development tools
- prototype run-time engineering