# ANSA Reference Manual

## Release 01.00

## Part VIII: Technology Projection

# Part VIII
# Chapter 1
# Implementation options

Editorial: The purpose of the technology projection is to explain how to use the architecture in the context of computers, networks and software systems defined outside the architecture, by showing the correspondences and conflicts between the architecture and available technology. In the present release of the Manual, all that has been done is to collect outline notes for the guidance of designers. In future releases, this guidance may be extended to include named implementations of the architecture, thereby defining architecture standards for 'physical' conformance in addition to the 'design' conformance requirements given for each projection. (Some authorities use the term 'compliance' to distinguish conformance to a design, rather than conformance which can be physically tested ).

## 1.1 Computational model

Editorial: the computational model is organized so that various forms of checking of a program can occur at different times. Notes for guidance in this section will relate the options for separating checks to conventional software tools: macro/pre-processors, syntax analyzers, code generators, linkers and loaders.

### 1.1.2 Programming languages

A designer need only model operations that can be invoked from one capsule upon another using the ANSA computational model, since only the behaviour, and not the structure of a remote capsule is exposed in interface type definitions. All of the interactions internal to a capsule can be implemented according to a different computational model provided that a mapping exists between some elements of the foreign computational model and the interface and operation invocation features of the ANSA computational model for the external interactions.

This gives the designer considerable discretion in the granularity at which the computational model is used. At one extreme, a capsule can be regarded as a single computational object and a programming language used within the capsule which does not conform to the ANSA computational model. Often the designer may be faced with an application which has already been written in a language that does not conform to the computational model and to treat it as a monolithic computational object is the only choice. At the other extreme, the designer may choose to program all the contents of a capsule as computational objects in a language which conforms to the

computational model. A midway position can also be taken in which some objects of internal significance are implemented as computational objects and others are not. From the point of view of interworking, these choices are indistinguishable: it does not matter from the outside what sort of language is used within a capsule. From the point of view of application portability the architecturally conformant implementation is more suitable, since the task of porting a program is merely one of transformation from one representation to another, rather than re-interpretation of one computational model in another. The former is a strictly mechanical operation; the latter is not necessarily so.

If a designer decides to implement some, or all, of a capsule in a language that does not conform to the computational model, some representation of external interactions must be chosen in the foreign computational model. There are two options. The first is give the application programmer direct access to an interpreter. This has three disadvantages:

- runtime checks must be made to ensure that the dynamic use of the interpreter functions is compatible with the constraints of the computational model (i.e. that library functions are called in a meaningful order and with reasonable arguments)
- low-level details such as network addressing and data encoding become exposed to application programmers, who become overwhelmed by the complexity of the interpreter
- applications programs become dependent upon the particular interpreter, which increases the difficulty of porting programs from one environment to another, and the difficulty of changing the mechanisms implemented by the library.

The second option is to extend the programming language used for writing local interactions with declarative rather than imperative formulations for the additional facilities required by the computational model: namely the means to specify interfaces (including transparency constraints) and to invoke operations. This approach has the advantage that the compiler can make static checks to ensure compatibility with the computational model for external interactions and generate an optimized interpreter automatically. The disadvantage is that the programmer has to know the details of how program statements in the extended language are mapped onto stubs and library functions when debugging programs, unless the debugging tools are also extended to understand the additional facilities. A benefit of this route is it makes programmers familiar with the computational model rather than the engineering model, and thereby facilitates a smoother transition into using a language that conforms to the computational model for future applications.

## 1.2 Engineering model

The conformance issue discussed in the engineering projection is interworking. The engineering model is an example of how to structure

systems to execute programs that conform to the computational model. The designer is free to select any system structure that has equivalent behaviour at the point of interconnection. The implementation notes contained in this section are specifically directed at the designer who chooses to follow the model in structuring an implementation.

### 1.2.1 Nucleus

The engineering model describes the nucleus as if each object within it has its own processing resources. Figure 1.1(a) shows the basic model from VII,3.

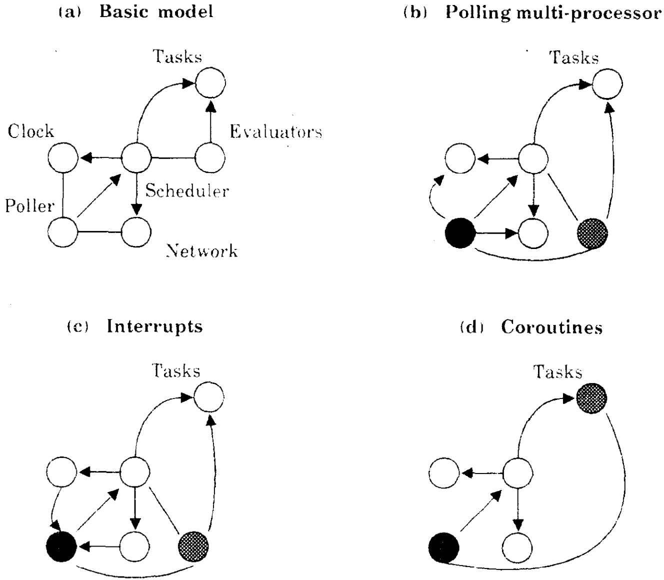
In many implementations there are not enough cpus available to assign them to individual objects. To this end the nucleus has been designed so that only one object within the nucleus need be processing at a time. The minimum requirement for a capsule is a single 'system' processor which will be multiplexed in time between the scheduler and other nucleus objects, including the evaluator objects that evaluate denotations. The 'system' processor may be a cpu in which case the nucleus is scheduling physical resources and can give performance guarantees. Alternatively, the 'system' processor may be a virtual cpu, provided by an underlying operating system. In the second case a capsule has to compete for the resources of the underlying computer against other virtual cpus and therefore it may be harder to give performance guarantees.

The implementation freedoms for the nucleus rest on the connections between the poller, the processors, the scheduler and the network.

Figure 1.1(b) shows an implementation of the engineering model using two cpus. One cpu, shown in black, is committed to the poller; it monitors (i.e. polls) the network and the clock for the arrival of data and clock ticks respectively. The other cpu, shown in grey, processes tasks. When either network or clock events occur, the monitor processor calls the scheduler **event** operation to process the event. When the control returns from the scheduler, the monitoring cpu signals to its partner to stop processing its current task and transfer control to the scheduler. Thus the inter-cpu signal is an implementation of preemptive use of the **transfer** operation. The **transfer** operation upon the evaluator is also invoked by the scheduler in response to the call of **schedule** (i.e. non-preemptively when a thread is blocked) and modifies the program counter and stack pointers of the cpu to pick up a new task. The program counter will be set to simulate a return from the call to **schedule** by which the thread associated with the new task last went idle. Since both cpus may simultaneously want to enter the scheduler it should be implemented as a monitor to avoid conflict.

Another way to achieve the multiplexing is to use interrupts to signal clock and network events. This is show in Figure 1.1(c): the poller is equated to a cpu in interrupt mode and the evaluator and all other objects to the cpu in normal model: these two modes partition the cpu in time. In such an implementation the poller would execute an interrupt handler which calls the **event** operation of the scheduler in response to an interrupt from the

**Figure 1.1: Nucleus implementation**




network or clock. On the occurrence of an interrupt the state of the current evaluation at that point must be recorded in a task, since the cpu is about to be re-used. Thus entry to the interrupt handler is the **preempt** operation and exit from it is the **schedule** operation invoiced by the processor object. When the cpu is in normal mode and interacting with the scheduler it will be necessary to inhibit interrupts so that there is not conflict with external events. The **transfer** operation in this implementation would be similar to the multi-processor case.

The alternative to an interrupt structure is to use coroutines: this is illustrated in Figure 1.1(d). Tasks are implemented as coroutines (shown as the grey object) driven by a coroutine coordinator (shown in black) which combines the functions of poller and evaluator. There is no risk of synchronization operations being interrupted in such an implementation, and so eventcounts and sequencers need not be atomic in this implementation style, nor need the scheduler be protected. However the network and clock will only be polled when control is returned from the coroutine coordinator to the scheduler. If this is not sufficient to guarantee real-time response, the coordinator must call the scheduler **event** operation in between scheduler interactions provoked by tasks. In a coroutine implementation, the **event** and **preempt** operations are null and the **transfer** operation is a coroutine entry primitive.

Thus, depending on the implementation style chosen, the mechanism for interaction between nucleus components may be variously mapped onto coroutines, subroutines or interrupts. The means by which objects in the same capsule can be given different invocation mechanisms is discussed in the next section.

Because the scheduler is permitted to assign any processor to any thread, and hence to any denotation, all of the processors available to it are required to be able to execute all of the denotations in a capsule. In practical terms this requires that all the processors must have an identical instruction set and share a common memory.

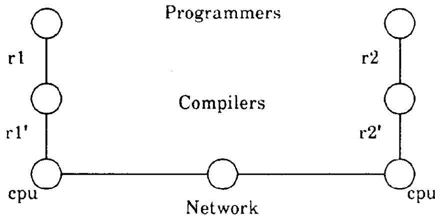
The implementation of the network used for communication between nucleus objects depends to some extent on the relationship between nucleus objects and physical computers. If each nucleus is in a separate computer, the network must consist of i/o devices interconnected in a star-shaped network with a central switching function, or perhaps a wide-area telecommunications network, or a local area network. If nucleus objects are in the same computer, the network may achieve communication by synchronized use of shared memory, via some form of inter-process message system, or via an inter-processor bus for example. A capsule which has access to two networks can act as a relay for taking data from one network and passing the data to another.

### 1.2.2 Interpreters, compilers, libraries and stubs

The computational model explains the function of an interpreter for computational objects as graph reductions. Consequently one strategy for implementing the computational model is to provide cpus which can simulate exactly the process of graphical reduction described in Part VI. This may be a hardware simulation, or a software simulation running on an ordinary computer. Simulators such as this are often called **interpreters.** Software simulators, especially for simple operations, can be inefficient and so an alternative strategy is to transform denotations into a form which can be executed directly by an ordinary computer. This form of translation is an example of **compilation**. A programmer submits a program representation to a compiler which transforms the representation and then submits it to a processor for execution. (In many systems, the compilation process is broken down into a chain of objects including pre-processors, translators, code generators, linkers and loaders; the details of the chain make little difference to the discussion in the section except to offer more points at which various of the techniques being discussed can be applied.)

The compilation process may be distributed, and the format of representations at different nodes may differ. This is shown in Figure 1.2(a). Two programmers are shown interacting with separate compilers for languages r1 and r2 which conform to the ANSA computational model. The compilers are connected to different processors, which are nodes of a network (i.e. they are in different capsules). Each compiler can transform the

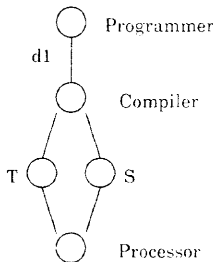
**Figure 1.2(a): Compilation**




program it is given into a form which can be interpreted by the cpu at its node (r1', r2' respectively).

Often the designer will have available processors which provide many, but not all, of the operations required to interpret computational objects, in which case the compiler can only translate some parts of the object representation directly into processor instructions. For ANSA, these instructions will be predominantly those that lead to nucleus operations. In this situation the designer has to provide an interpreter for the missing functionality. This structure is illustrated in Figure 1.2(b) where the compiler output is shown as consisting of two parts: one is that corresponding to the directly translated parts of the representation (T) and the second is that which is directed to the interpreter (S).

**Figure 1.2(b): Interpreter emulation**




There are three ways in which the interpreter may be implemented.

The first is simply to substitute the simulations inline when they are needed. This is effectively **macro-processing** r1, with macros for those parts of r1 that correspond to S. This can be expensive in memory resources, especially in the case where a great deal has to be emulated; there is however no time overhead.

The second strategy is to reserve some instructions from the cpu instruction set to transfer between T and S and to provide a runtime interpreter for S. The disadvantage of this strategy is the runtime overhead of decoding S. The advantage is that it may be possible to encode S compactly and save memory compared to inline expansion. Also the interpreter can be separately linked to the application program, allowing both to be developed and maintained independently.

The third strategy is to combine the previous two approaches: S is encoded as operation invocations in the native instruction set and a library of **stub** objects is generated from S to implement each required simulation in terms of operations in T. This approach has significant benefits if S contains lots of common elements. For example, all of the invocations to an interface can share the same stub for marshalling and unmarshalling. In effect the library of stubs is a specific interpreter for the emulated functions required by r1, rather than a general interpreter with all the missing computational model functionality within it. The stub approach pays the cost of decoding at compile time rather than at runtime.

With this approach, different stub implementations can be substituted for different objects, taking advantage of access transparency. For example, stubs can map different nucleus operations onto coroutine calls, software interrupts or subroutine calls according to the chosen implementation style.

From the architectural point of view these three strategies are they same in function; the differ only in which parts of the function are performed in which epoch. The designer should therefore select the technique, or combination of techniques, that fits best with the compilation systems and computers he has to use. Designers for different systems can make different selections, since the internal structure of each system has no impact on their ability to interwork: interworking is defined with reference only to the communications between systems.