5. A Prototype USS

Chapter 5

A Prototype USS

"God help us; we're in the hands of engineers."

Ian Malcolm, Jurassic Park

This chapter presents the implementation of a prototype USS based on the design described in the previous chapter. A full implementation of the design would take a considerable amount of time, far in excess of that available to the author. Therefore only those components (or parts thereof) that were required to demonstrate the architecture's key points were implemented.

The feasibility of implementing a scheduler on top of a general-purpose operating system is explored with the implementation of a solution to the real-time VE displays problem. This is followed by a description of the platforms upon which the prototype was designed to run.

The USS implementation details begin with an examination of networking in a heterogeneous network, proceeded by configuration control and an implementation of a UML interpreter. Following details on each system component, the chapter concludes with a list of improvements that can be made to the prototype.

5.1 Real-Time in the Real World

The QNX operating system (QNX Software Systems Ltd., Ontario) was used by the author to develop the VE Support System (VESS) for experimental work undertaken in the VEL, University of Edinburgh. QNX is a Portable Operating System Interface (POSIX[1]) compliant, multi-tasking, distributed, real-time operating system (OS). It provides a priority-driven, preemptive scheduler which is certainly suitable for a soft real-time system and with great care can be used in a system with static hard real-time constraints. Part of VESS's functionality was enforcing the constant update rate of the CIG displays. The implementation of this solution is presented in this section and was used to explore the viability of implementing a scheduler-based prototype USS.

[1] The 'X' would appear to have been added to reflect the fact that the interface is based heavily upon the UNIX variants.

Section 4.3 presented a taxonomy of real-time scheduling algorithms. In the field of VR, many systems claim to be real-time and can indeed be classified as *soft* real-time systems. The service degradation option for ensuring a constant VE display update rate discussed in section 3.3.3.1 requires a deadline scheduler. Unfortunately, implementing such a scheduler on top of a normal multi-tasking OS such as UNIX is problematic. Most OSs are not suited to real-time purposes, i.e. they do not provide ways of guaranteeing response times for certain events such as interrupts, IPC and disk I/O, etc. Those real-time systems that do provide such guarantees often use static schedulers. A VE system is dynamic and therefore a scheduler is required that can *also* cope with changing existing deadlines and the introduction/removal of new tasks. Since a dynamic deadline scheduler was not available it was decided to adopt the worst-case operation solution (section 3.3.3.2).

5.1.1 Real-Time Displays

There are a number of operations and pieces of information that a visuals manager needs to enforce a fixed frame rate in the CIG:

1. Manual control over buffer swapping
2. The time between one display refresh cycle and the next.
3. The amount of time that the rest of the system components need to complete their work for the next simulation update.

5.1.1.1 Manual buffer swapping

This is essential to the task at hand. Double-buffered systems will display the last rendered image until the current one has been finished. At this point the new image is displayed and the next image is rendered into the other buffer. The switch actually happens during the *next* vertical retrace (or flyback) phase. On displays such as monitors, this is when the electron gun makes its way from the bottom-right corner of the tube (as the viewer sees it) to the top-left, ready to start drawing the next picture. To achieve a constant frame rate we must be able to *choose* which vertical retrace is used to switch display buffers.

5.1.1.2 Inter Refresh Time (IRT)

The IRT is the time it takes to draw one picture on the display including the vertical retrace period. For example, say that a 640x480 resolution image is refreshed at 60 Hz. This means that the IRT is 1000 / 60 = 16.66 ms. The refresh rate varies depending on the resolution of the video signal, e.g. an 800x600 pixel image is often refreshed at 72 Hz, and different display devices can handle different ranges of refresh rates.

The refresh rate may be provided as a parameter at run-time or, alternatively, this information may be obtained from the CIG which is the approach adopted here. Each time the CIG generates a vertical retrace it also generates an interrupt which is intercepted by the host machine and the time stored. The next time an interrupt is caught, the time difference is calculated and this gives us the IRT.

This technique will only work if the host machine has a clock that can provide nanosecond accuracy and the interrupt latency[2] is bounded. The latter point is by no means certain in non-real-time operating systems such as UNIX and was one of the main reasons QNX was used.

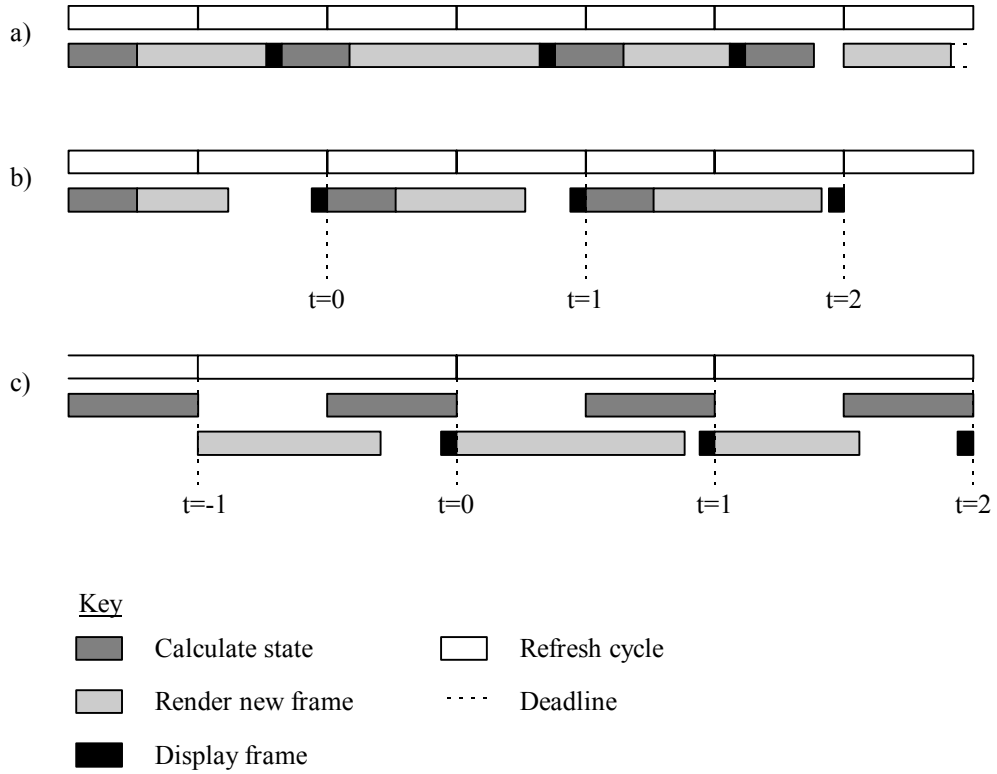


Figure 5.1 Simulation cycle scheduling.
a) buffer swaps happen at unpredictable times during the next simulation cycle in a variable-rate system; b) controlled buffer swapping in a single CPU fixed-rate system; c) a multiprocessor fixed-rate system permits the calculation stage to be done in parallel and in advance of the rendering stage resulting in a faster update rate.

5.1.1.3 Inter Update Time (IUT)

The total processing time required for one simulation update is provided by the scheduler and the IUT is the nearest multiple of the IRT to the given time. In other words, the total work time can be expressed as a number of display refreshes. For example, if the IRT is 16.66 ms and the work takes 40 ms, the IUT would be 49.99 ms, i.e. the work may be done within 3 refreshes of the display.

[2]The time between the interrupt being generated and the process on the host machine being notified of the event.

5.1.1.4 A comparison of paradigms

Figure 5.1 shows the various ways of scheduling the work to be done each frame. There are three basic stages: *calculate*, *render* and *display*. Figure 5.1a shows how these stages fit together in a variable-rate system and how they relate to the display refresh cycle. The time at which the frame may be displayed varies and rarely coincides with a vertical retrace, which means that the actual buffer swap happens sometime during the next cycle. As shown in the diagram, most of the time the calculation stage may progress immediately and by the time this is finished, the buffers have been swapped and the render stage is ready to continue. However, the last complete cycle in Figure 5.1a shows that it may be necessary for the render stage to *wait* until the buffers have been swapped. This is because the buffer that will be filled next is currently being displayed.

The scheduling of the work in a fixed frame rate system is shown in Figure 5.1b. The time between the end of the rendering stage and the display will vary depending on how long it takes to render the scene. Pseudo-code for this process is given in Figure 5.2.

```
// Step 1: Initialise key variables

Calculate IRT
Calculate IUT based on totalWorkTime
Enable manual buffer swapping
displayTime = 0


// Step 2: Synchronise loop with display

Wait for refresh


// Step 3: Enter main processing cycle

While simulation not complete
{
     // Step 3.1: Calculate state

     displayTime = displayTime + IUT
     Calculate state of VE for displayTime


     // Step 3.2: Draw new image but don't display

     Redraw display


     // Step 3.2: Display image exactly on time

     Wait for end of IUT period
     Swap buffers
}
```

Figure 5.2 Pseudo-code for the fixed frame rate, worst-case simulation cycle.

Both these examples assume that all work is being done by one CPU. If the image generation can be dedicated to another CPU or the system is equipped with a separate graphics subsystem, then time may be saved by scheduling the calculate and render stages such that they overlap as shown in Figure 5.1c. This is best achieved by starting the redraw as soon as

possible (since it will take the longest time to complete). In order that we are rendering the most up-to-date state possible, the calculation stage is done before the end of the previous frame. By performing these two stages in parallel it also means that more time can be spent on the simulation dynamics. Obviously, failure to complete either of these stages before the designated refresh occurs is a system failure.

Regardless of technique, it is important to understand how the CIG works and the latency that it introduces into the process since not all CIGs work the same way. For example, an SGI RealityEngine/2™ introduces a one frame latency whilst the Real World Simulation Reality3™ PC card produces a two frame latency. The latter system was used in this implementation and, to compensate for this latency, state calculations must be done *two* updates *before* the image needs to be displayed.

This method of controlling double-buffering can be applied to most CIGs with few problems since it utilises existing functionality. It may be necessary, however, for the API to be modified to gain access to this functionality.

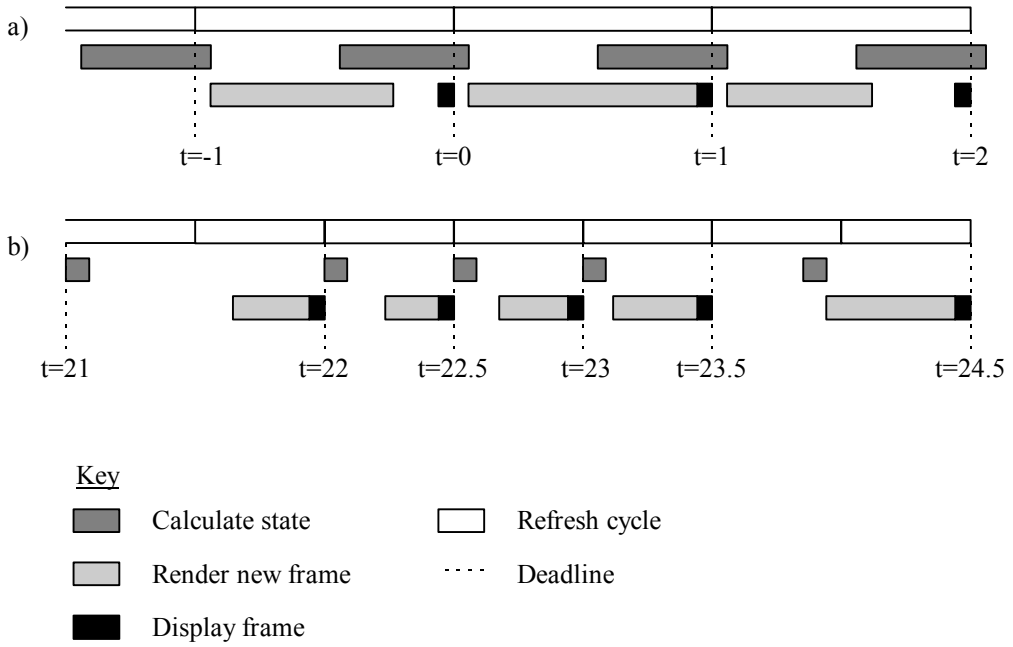


Figure 5.3 Improved simulation cycle scheduling.

5.1.1.5 Further improvements

It is quite common for the render stage (even in its worst-case) to complete before the time that the display stage needs to run (as shown in Figure 5.1c). If this is the case then the start of the state calculation, which includes input device sampling and the render stage, may be put back such that there is even less delay between calculation and display (Figure 5.3a).

A more advanced technique is the controlled increase or decrease of update rate. It would be possible to detect whether the CIG is capable of going faster, e.g. making the change between 30 Hz and 60 Hz, by maintaining a history of its execution time for each update. If, after a small period of time, this new potential performance was sustained then the other stages could be rescheduled, if possible, and the switch made (Figure 5.3b). In a similar way, by monitoring the performance profile, a slow increase in workload could be detected and a

decision made to extend the deadline. Once a decision is taken to change the deadline, no further changes must be made for a reasonable period of time, e.g. a couple of seconds, or things would quickly degenerate into a variable-rate system. Such an enhancement could also help overcome the fact that the worst-case approach assumes that the environment is quite static and does not cope well with the dynamic creation or destruction of objects.

> a) shifting the calculate and render stages to reduce system latency; b) performance profiling permits the increase/decrease of the update rate in a controlled manner.

Some multiprocessor CIGs already monitor image complexity to aid in processor load balancing. For example, the Reality3™ system, uses knowledge of the changing complexity of each scan-line to predict the load distribution for the next update. With additional functionality in the API, these calculations could be used in the decision-making process. It is true that simple decision-making logic could be flawed by fast increases or decreases in workload, but the potential increase in system fidelity makes it worthy of more investigation.

A deadline-based approach also provides the framework for the application of object priority systems within the CIG as well as the visuals manager. Objects may be drawn, partially drawn or skipped depending on their priority (as in Holloway's Viper system).

5.1.2 Conclusions

The problem of presenting a temporally correct view of a VE has implications throughout the whole support system architecture. The most important (and often the most expensive) component of a VE system is the CIG. Most CIGs provide some kind of service degradation in the form of LOD (section 3.3.3.1), but this is insufficient and improvements must be implemented via the API.

The implementation presented above has been used effectively over a number of years in the VEL. However, its utilisation is not as simple as "plug and play" since its performance is highly dependent on the other processes used to simulate the VE. For example, if data logging is added to the simulation then this introduces an execution path that passes through the filing system manager and the hard disk device driver. Each of these processes have their own timing constraints, are dependent on a number of interrupts and must therefore be accounted for in the schedule. Other changes that can have large effects on reliability are: communicating with a machine via the (dedicated) network, increasing the complexity of the visual database being used, adding another input device, synchronising with an external device, etc.

Even under QNX, which supports POSIX 1003.1b Real-Time Draft Standard Process Scheduling, getting an application to schedule every component to meet worst-case deadlines can be quite time consuming. The possibility of doing the same under a heavyweight OS such as System V Release 4 UNIX is very low. In addition, general OSs use virtual memory and have unbounded interrupt latency to name but two confounding features. Since it was the intention to demonstrate USS running on different machines and operating systems (albeit UNIX variants), it was decided not to attempt a real-time implementation.

5.2 Target Platforms

From the outset it was intended that the prototype should be portable to a number of different platforms. It was planned to use QNX during initial development; so it was a natural progression to use other platforms with similar operating system functionality, preferably with some POSIX compliance. These platforms are briefly described in this section whilst specific details are dealt with in section 6.2. The choice of an Implementation Language, IL, is also discussed.

5.2.1 IBM Personal Computer Compatibles

Three PC compatibles on a dedicated network within the VEL were available to the author, each running QNX. One of these machines acted as a gateway to the Internet thus opening up the possibility of connecting multiple USSs on a heterogeneous network. Each machine had between 16 and 24 Mbytes of main memory and ranged in power from an Intel 486/50 MHz to an Intel Pentium/90 MHz. The memory capacity is important because QNX does not use virtual memory. Additional resources included a dedicated CIG and sound generation equipment.

5.2.2 Cray T3D

Originally it was intended to use the Edinburgh Parallel Computing Centre's (EPCC) Cray T3D super-computer as the second platform to run the prototype. The T3D was installed with 160 nodes, each with 2 DEC Alpha 21064 processors running at 150 MHz and 128 Mbytes of memory (64 per processor). The T3D is connected to the real world via a Cray Y-MP host running UNICOS, a POSIX compliant OS. Unfortunately use of the Cray had to be abandoned for a number of reasons:

1. Despite having an 8 MByte "microkernel", no IPC mechanism is provided - only shared memory operations are available. To ease this problem, three messaging libraries are available:

    a) Portable Virtual Machine (PVM - Geist & Sunderam, 1991). This library makes use of a central server process which runs on the T3D host. Unfortunately the central server process does not fit with the USS design.

    b) Message Passing Interface (MPI, 1993). This is an attempt to standardise on an IPC mechanism incorporating features of many such libraries, including PVM. However, it is very rigid and imposes requirements on how the programs must be structured that conflict with USS design.

    c) Fast Messaging (FM - Karamcheti and Chien, 1994). This unsupported library provides a low-level IPC mechanism using shared memory routines which provides latency an order of magnitude lower than PVM. This would be the library of choice but even this could not overcome the other problems detailed below.

2. A process runs on one physical processor. There is no multi-threading support and this can only be achieved by using a large conditional statement in a monolithic program to select alternative execution paths. To port a multi-process system to a one process per processor architecture would have involved major changes and be grossly inadequate. The other alternative would be to have one system component running on each processor and treat the whole machine as one node. This would, of course, be absurdly inefficient since many processes, such as entities, are inactive for a large proportion of their life.

3. The Cray C++ compiler does not support exceptions which were used extensively in the prototype (section 5.2.5). Removing exception handling code from a program requires a total re-design and re-write.

4. Whilst it was possible to communicate from the T3D to the outside world through the Y-MP host using a "message-routing" process, the author was advised against trying. The host was so heavily used any such routing process would have to wait a long time to gain access to the CPU thus shattering any hope of reasonable real-time performance.

5.2.3 Sun SPARCcenter

The Sun SPARCcenter 1000E met all of the required criterion and was used to develop the prototype in parallel with the QNX version. SunOS v5.4 supports some of the POSIX standards which made porting relatively straight forward. However, this machine is used by many in the University as a compute server and therefore could not be used to evaluate system performance.

5.2.4 SGI RealityStation

A network of three SGIs arrived in the Department of Computer Science half way through the final year of this project. The most powerful of the machines was a RealityStation which is populated with 128 Mbytes of main memory and runs the IRIX OS (v5.3) which uses virtual memory. Unfortunately a suitable C++ compiler was not installed until a couple of months before submission, limiting work on this platform to a minimum.

5.2.5 Implementation Language

Development of the prototype started under QNX which supported ANSI C and C++ with exceptions and templates. In general, the code generated by a C++ compiler is as efficient as a C compiler and since object-oriented techniques lend themselves well to the task at hand, C++ was chosen as the implementation language. The availability of templates eased development and exception handling helped produce an easier to understand implementation. Watcom C++ v9.52 was used under QNX and Sun Professional C++ v3.0.1 was used to initially develop the prototype under SunOS. Later, compatibility with GNU C++ v2.7.0 was tested as a precursor to the SGI port and to aid debugging (section 6.2.1).

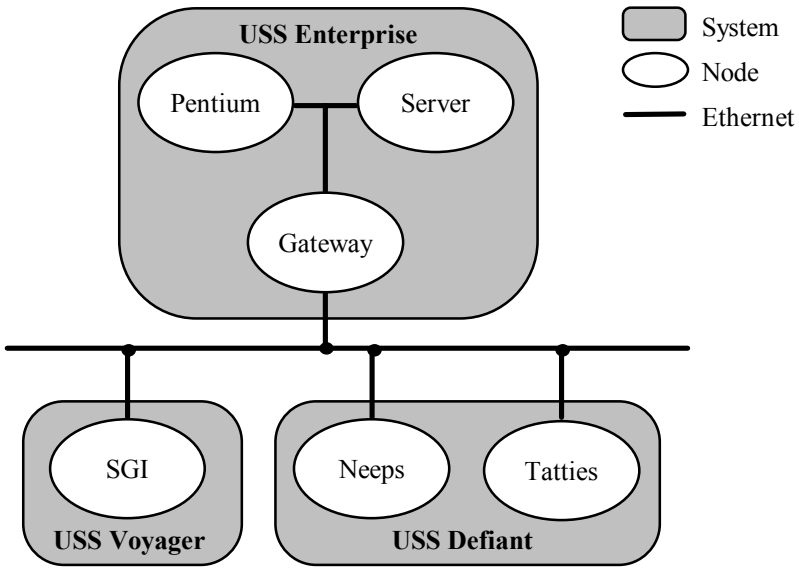


Figure 5.4 Example network configurations of three USSs.

5.3 Networking

The actual organisation of USSs need bear no relation to the physical location of the nodes or their internetworking. Figure 5.4 shows three possible configurations of a USS, all of which are connected to the same backbone network. System *Enterprise* is constructed from three nodes interconnected by a dedicated network with one node acting as a gateway to the backbone. System *Voyager* only has one node whilst *Defiant* has two nodes but its local communications must share the bandwidth with all of the other traffic on the backbone. Whilst this last configuration is not efficient, it is functionally valid.

There is no required medium or protocol for interconnecting systems. In this example, however, all the nodes use Ethernet as their communications medium. It is possible that the medium used within USS Enterprise could be totally different provided that an Ethernet link to the other systems was still maintained. This would be the situation in a multiprocessor system where each processor could correspond to a node.

Each OS has its own mechanism for sending messages to processes within its domain of control. On a single processor system this means sending messages between logical processes running on the same processor and may be implemented as either sharing or copying memory. This is also true in some multiprocessor systems where memory is shared, in others communications may use high-speed links between processors. In distributed systems the message may also be sent between physical machines over a high-speed LAN connection. The one criterion that links all these different domains is that the recipient is directly addressable by the operating system.

5.3.1 IPC Mechanisms

Most operating systems provide their own method of lightweight message-passing, e.g. QNX, but others rely on more heavyweight methods such as TCP/IP, e.g. IRIX. Under QNX, multiple machines may be networked together into one virtual machine and the system's IPC mechanism works between processes on different nodes as if they were on the same physical machine. It can coexist in an Ethernet network with other protocols but cannot be used to communicate with systems that are not running QNX. In order to communicate with processes outside the native domain of control it is necessary to use a different delivery system, such as TCP/IP. This also means that a different addressing method must be used.

To localise the impact of these differences (and those of other OSs), a *Process Management Layer* (PML) is incorporated into each system component which sits in between the operating system and the component implementation (Figure 5.5). This process layer provides a set of services (presently just IPC) which are independent of the underlying operating system. Where more than one delivery system is available the layer chooses the right mechanism for the right job. How these decisions are made is platform and implementation specific. There is only one requirement, of course: the message delivery must be *reliable*. The prototype supports QNX IPC, TCP/IP and the framework for supporting a shared memory IPC mechanism is present but not fully implemented. UNIX domain sockets (which are faster) were not used instead of TCP/IP because QNX does not support them and they would complicate system performance comparisons (section 6.4).

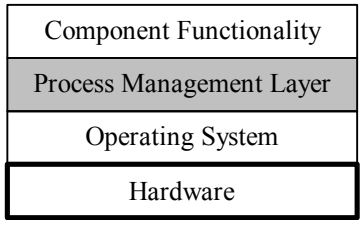


Figure 5.5 Position of the Process Management Layer within the system software.

5.3.2 Addresses

Each process within the system has an address which is unique throughout all USSs. The address is made up of three components: the *system ID* (SID), the *node ID* (NID) within that system and the *process ID* (PID) within that node (Table 5.1).

Current sizes are signed 16 bit integers for both the SID and NID, with an unsigned 32 bit integer allocated for the PID. Valid SIDs and NIDs are positive integers - negative values are used during the process' initialisation phase. This provides a unique address for 32768 systems, each with up to 32768 nodes, each of which may have 2^{32} processes running on them. This is truly overkill for the prototype but offers a realistic address range when large-scale distribution is a goal.

USS ID	USN ID	Process ID
16 bits (signed)	16 bits (signed)	32 bits (unsigned)

Table 5.1 Message address structure.

When each system is defined in the systems' configuration file (section 5.4), it is allocated a unique SID. Likewise, each USS definition contains a number of USN definitions which specify a NID that is unique within that system. The PID is different because the number used is unique within the given node. It is used to reference the process that the message is intended for (or sent by), but how it is used to locate the relevant process is implementation and thus node dependent. When using QNX IPC, messages are indeed addressed using the operating system's process identifier, whereas an implementation using TCP/IP uses the socket number associated with the process. A shared memory implementation would use the address of the memory block holding the message queue.

5.3.3 Messages

All communications between the components of the USS use a number of pre-defined messages whose basic structure is shown in Table 5.2.

From	To	Message ID†	Transport ID†	Length	Data
8 bytes	8 bytes	1 byte	1 byte	4 bytes	optional

† Aligned on a 2 byte boundary, i.e. requires one padding byte.

Table 5.2 Message header structure.

The address of the sender and the intended recipient are the first two fields in the message header. The recipient field is necessary because the message may be routed through one or more other processes before it arrives at its destination. The message ID number is used by all system components to determine whether to deal with the message and, if so, how to decode the data (if there is any). The desired method of transportation to the recipient is also recorded in the message.

The size of the associated message data is given in bytes. The interpretation of the data depends on the message ID. A list of the defined message types and their purpose is given in Table 5.4. Message IDs are often reused for slightly different purposes, the exact meaning depending on the receiver, e.g. entity, manager, etc. In addition, many messages share the same physical structure with regards to data contents (Table 5.3). For example, all messages that contain UML information (binary or ASCII) use the same structure: 7.

Type	Name	Size	Description
-1	-	0	All information required is in the message header.
0	String	*n*	Used to send variable length textual information.
1	Notify	4	Holds reason for process termination.
2	Ping	28	Holds flag indicating whether receiver issued ping or is being pinged and timestamp information.
3	Profile	16+	Holds a variable length RP.
4	UPID	48	Room for both the name and UPID of a process.
5	UPID2	16	Contains just two unnamed UPIDs.
6	Status	4	Details the status of a previously requested service.
7	UML	24+	Holds either an ASCII UML definition or binary state data.

Table 5.3 Description of the nine physical message structures.

5.3.4 Hardware Differences

Sending messages between machines in a homogeneous environment requires no additional effort. However, in a heterogeneous network there are hardware architecture differences.

5.3.4.1 Byte Order

The byte ordering used in CPUs may be classed as either little-endian or big-endian. A little-endian CPU, such as those produced by Intel, places the least significant byte of a word first. Conversely, a big-endian CPU places the most significant byte first. The reasons behind the choice of one ordering over another will not be discussed here but recently some CPUs have been built such that the byte ordering used can be selected by setting a bit in one of the CPU's registers, e.g. Motorola 88110 (Motorola, 1992).

5.3.4.2 Floating-Point Representation

5.3.4.3 Another difference may be the representation of floating-point numbers: single-precision (32 bit), double-precision (64 bit) and extended precision (64 bit and upwards). This is less of a problem since most general-purpose CPUs conform to IEEE 854 (IEEE, 1987) although they may, of course, have a different byte order.

5.3.4.4 Memory Alignment

Some architectures also require certain data types to be aligned on given byte boundaries. For example, a 32 bit integer may have to start on a 4 byte boundary. If not required then often operations are performed more efficiently if aligned on these boundaries. In these cases the alignment is enforced by the compiler or provided as an option (Watcom, 1995).

Message	Type	Purpose
GET_UPID	4	Sent to the UM to obtain the sender's UPID.
SET_UPID	4	Sent by the UM, containing the recipient's UPID.
PING	2	Test connection/measure round-trip time to a given process.
NOTIFY	1	Inform the UM why this process is terminating.
RPROFILE_NOTIFICATION	3	Holds a process or node RProfile.
RPROFILE_REQUEST	3	Sent by a process wanting a process or node RProfile.
LOCATE_REQ	0	Ask the UM to locate a process based on the specified search criterion.
LOCATE_RESP	4	UPID of the located process returned by the UM.
STATUS	6	Success/reason for failure of the specified message.
ACTIVATE_UM	-1	Notify the MUM that this node is active.
DEACTIVATE_UM	-1	Notify the MUM that this node is disconnecting/ tell slave node to terminate.
ACTIVATE_USS	-1	Notify the master USS that this system is active.
DEACTIVATE_USS	-1	Notify the master USS that this system is disconnecting/tell slave system to terminate.
TERMINATE	-1	Sent by UMs to force termination of any given process.
CREATE_ENT	0	Execute the given process on the recipient UM's node.
CREATE_ENT_ACK	-1	Sent by SUM to MUM when an entity has been created.
DESTROY_ENT	0	Terminate the given process on the recipient UM's node.
DESTROY_ENT_ACK	-1	Sent by SUM to MUM when an entity has been destroyed.
UML	7	Holds valid UML code to be parsed by the recipient.
UML_INIT	-1	Request the sender's UML definition from the UM.
UML_INIT_DEF	7	New, complete UML definition sent by the UM.
UML_CONSTRUCT	7	Execute entity's `Construct` function/entity's initial state information.
UML_UPDATE	7	Execute entity's `Update` function/send state updates.
UML_DESTRUCT	-1	Execute entity's `Destruct` function.
UML_MONITOR	0	Manager's registration of interest in part of the UML definition.
UML_MONITOR_ACK	-1	Sent by the UM to confirm acception of a monitor request and inform entities of dependency.
UML_SYNC	0	Request current list of UML dependencies.
UML_UPDATE_NOTIFY	-1	Notify entities that they should update and managers that they should expect UML_UPDATE messages.
UML_UPDATE_COMPLETE	-1	Notify managers that all entities have updated.
MIGRATION_NOTIFICATION	5	Informs receiver that a migration has occured - contains the process' old and new addresses.
MIGRATION_REQUEST	4	Sent by a RM to the MUM to request an entity migration.
MIGRATION_STATE_REQ	-1	Sent to an entity to obtain a complete copy of its state.
MIGRATION_STATE	7	Complete entity state sent from source to target entity.
MIGRATION_STATE_ACK	-1	Used to inform the MUM that state transfer was successful.

Table 5.4 Summary of message types and their use.

5.3.4.5 Transfer Format

The External Data Representation (XDR) library of functions are used to represent data structures in a machine-independent form (Bloomer, 1992). This library is available on most machines running UNIX and can be used to encode dynamic data structures as well as just handling the primitive types. Due to this level of functionality it is also quite a bulky library with respect to both memory requirements and the API. Even the low-level code used by Snowdon (1995) produced a significant overhead.

Of the platforms available for use by the author, two used big-endian ordering, one used little-endian and all of them used the same single and double-precision floating-point formats. Since the UML data structure traversal routines had already been written and the number of messages types sent between machines was relatively low, it was decided to provide hand-coded byte-swapping routines. In addition, although XDR is a popular library, it may not be available on all systems which would cause problems porting USS.

The chosen format for sending messages was little-endian because the big-endian machines had more powerful CPUs and could better accommodate the overheads involved in encoding/decoding. The byte-swapping code was conditionally compiled into big-endian systems to minimise code size and maximise execution speed on little-endian machines. As the process layer receives messages, it encodes/decodes those that are destined for/received from other nodes.

5.3.5 Layer Implementation

Each process in a USS is both a provider and a consumer of services. A service is requested by sending a message to the provider which performs some processing and then possibly sends a result back to the consumer. Information flows between processes freely and it is possible for two processes to be each other's consumers and providers. The PML provides the nuts and bolts that can support this functionality and avoid deadlock.

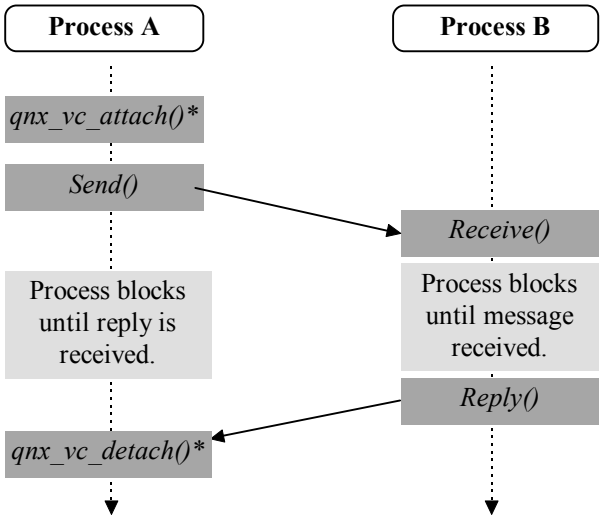
5.3.5.1 Asynchronous

Synchronous message transmission is a convenient mechanism for issuing service requests but can leave the sender waiting for a response when it could be doing other work. In USS, therefore, all processes send a message and then continue immediately with other processing. Some time in the future they may receive a response to their original request which must be associated with it in some way. This may be explicit by including a reference in the response or implicitly because it could only have come from one message.

5.3.5.2 QNX

Messages are sent between QNX processes using a three stage procedure: *Send-Receive-Reply*. Figure 5.6 shows the sequence of these stages and what happens to the state of each process. After a message has been sent, the sending process blocks until it receives a reply from the message's recipient. Similarly, when a process enters the receive state it blocks until it is sent a message at which point it can do some processing and then must issue a reply. It is possible to poll for a message but continuous use of this service will seriously degrade system

performance. To minimise the time that the sender is blocked, a reply is issued immediately after receiving the message.



* Only when sending to another node.

Figure 5.6 Send-Receive-Reply procedure for sending messages under QNX.

Sending a message to another node in a QNX network requires the establishment of a *virtual circuit* between the sender and receiver. The identifier assigned to this circuit is then used when sending the message instead of the PID in the message address. After the reply has been received the virtual circuit is deleted. It would be more efficient to leave the virtual circuit in place and re-use it the next time - an operation supported by QNX. However, the burden placed on the operating system by the potentially large number of circuits could degrade system performance. The buffer used for sending messages within the operating system grows as needed but it is also possible to send multi-part messages which keeps the required buffer size at a minimum.

5.3.5.3 TCP/IP

Each process obtains a socket number which is used throughout its lifetime as the PID component of the UPID. Whilst the contents of the PID field in the message address is enough to send a message under QNX, TCP/IP also requires a hostname to establish a socket connection. If the recipient is on the same node then the node's hostname can be obtained from the operating system. Any message destined for another node is sent through the UM which maintains a routing table[3] for each node in the system. If it is the MUM then it also stores a route for its counterpart in each system. A table entry is composed of the SID, NID and hostname.

The sequence of events required to send a message using TCP/IP as implemented in the process layer is shown in Figure 5.7. TCP/IP requires a connection to be established before

[3] Stored in and administered by the PML.

data transfer may commence. A similar phase is the creation of virtual circuits in QNX, but whereas QNX provides OS support for maintaining virtual circuits, it is up to the application to keep track of established socket connections. Each connection has to be periodically polled to check for incoming messages compared to issuing a single call to `Receive()`[4]. Since this would introduce unwanted complexity and a considerable overhead in the prototype, socket connections are established and closed each time a message is sent.

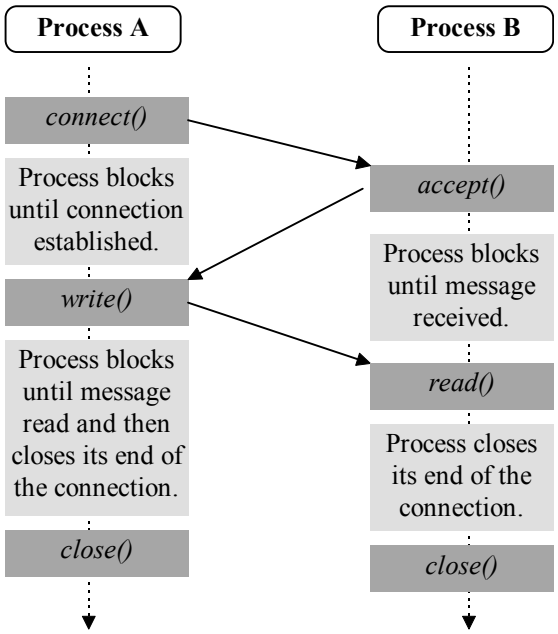


Figure 5.7 Message transmission sequence using TCP/IP.

5.3.5.4 Deadlock

A problem common to both of these implementations is that of deadlock. If process A should send a message to B at the same time as B sends a message to A then both will be blocked waiting for the other to receive the message. A solution is to split the layer into two processes. The first process holds all the components functionality and receives messages as per normal. When it wishes to send a message, it is passed to the second child process which actually sends it. Therefore only the child process ever becomes send-blocked leaving the parent process to accept incoming service requests and perform its usual work (Figure 5.8).

The overheads of this solution can be minimised by using threads (lightweight processes) which share both code and data, with a separate stack (Milenkovic, 1992). Messages could then be passed from parent to child by exchanging memory pointers. Unfortunately threads have not been implemented on all of the chosen platforms. A beta version of a threads library was available in QNX but was found by the author to be unreliable and so this option was ruled out.

[4] These overheads would not be incurred if an unreliable datagram (connectionless) mechanism were used.

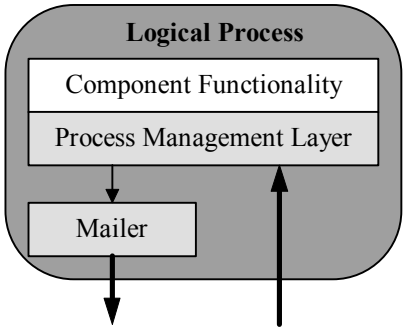


Figure 5.8 Structure of a logical process consisting of two physical processes.

The ability to create a child process using `fork()` is a common feature in UNIX-based systems. The child is, in effect, a duplicate of the parent process, sharing code but taking a separate copy of the data and stack. Although not strictly an IPC mechanism, *pipes* are commonly used to send data between two processes on UNIX-based systems. Pipes fall under the jurisdiction of the filing system but that does not require them to occupy disk space and may reside totally in memory. Since both these features were available on the target platforms this method of implementation was chosen. To reduce the often considerable memory overheads that `fork()` produces through duplication of data and stack, the child process, once created, is replaced by a lightweight *mailer*. This program simply reads messages from the pipe and sends them to their intended destination.

5.3.5.5 Initialisation

The PML is the first software element to be initialised when a process is created. Its first task is to determine the UPID of the process it is executing in. If it is a UM then initialisation is temporarily paused whilst the configuration file is parsed and then restarted when the node's SID and NID are known (section 5.4). The PID of the UM is the actual process identifier under QNX or a pre-defined port when using TCP/IP, i.e. 34000.

If the process is not a UM then it must locate its UM and send it a `GET_UPID` message. Location of the UM using TCP/IP is simply a case of connecting to the pre-defined port address. Under QNX the operating system's name server is used to locate the process identifier of the UM using a pre-defined name.

Upon reception, the UM allocates a UPID and returns it in a `SET_UPID` message which is subsequently processed and thus completes the layer initialisation.

5.3.5.6 Multiple Mechanisms

The layer can be initialised to handle both QNX and TCP/IP IPC. If so, connections on each mechanism are polled for, in turn, until one is established. This is a CPU intensive procedure if done continuously, but it is commonplace for each component to poll once for any messages before continuing with the outstanding work (section 0). Consequently, multiple mechanisms may be handled with only slightly more overhead than just one.

When there is a choice of methods for communication, the mechanism specified in the message is used. If this is left undefined then the best choice is used - the prototype will use QNX IPC in preference of TCP/IP. It is, however, uncommon for a message's transportation ID to be left blank, since it is accepted practice to respond using the same method that the request was sent with.

5.3.6 Networking Summary

In order to simplify the transfer of messages between processes and facilitate porting to different platforms, each software component has a process management layer. The interface to this layer, the message format and message addressing are the same regardless of the OS. In a heterogeneous environment, a common binary format must be agreed upon to enable machines with different hardware architectures to communicate. In the present day, these differences are far fewer and a compromise was found quite easily. As messages are sent they are encoded into the common format (if necessary) and decoded upon receipt (if necessary). To avoid deadlock the PML requires two processes to be used per logical process: one with the component-specific functionality and a small mailer process used to send messages. The PML's first action during initialisation is to ascertain its UPID, either through a configuration file or by communication with the node's UM. Once initialised, the network of PMLs can handle message transmission between nodes using different IPC mechanisms.

5.4 Configuration Control

Some of the components in a USS need configuration information when they are created. This section presents a simple language that is used to help fulfil this task and is followed by an example of its application: system configuration.

5.4.1 Universal Configuration Language (UCL)

This minimalist language provides a way of structuring simple information in a hierarchical manner. UCL is used by those processes that need configuration information upon creation. The UCL parser constructs a small internal data structure which may be read, manipulated by the process and then output again. Currently, this information is stored in files which are read by each process but there is no reason why this information could not be sent by the UM.

The basic building blocks of UCL are *Components* and *Variables*. A variable is given a type of *Real*, *Integer*, *String* or *Boolean* and lists may have mixed types. Every variable is required to have a value, but if this is not needed an empty string may be specified (`""`). A component can contain variables and zero or more other components which form a hierarchy, of which there may be many in each file. Figure 5.9 shows a contrived example of a UCL description that contains one of each possible construct.

```
Container containerName
{
    SubContainer componentName
    {
        aString   "hello"
        aReal     1.0
        anInteger 2
        aBoolean  FALSE
    }

    mixedList  1, 2.0, TRUE, "goodbye"
}
```

Figure 5.9 The basic elements of UCL.

Components are identified by a *type* name which is followed by an optional *name* that can be used for reference purposes during parsing and when accessing the information described therein.

UCL permits structuring of non-complex data in which ever way is most suitable for the task at hand. In order for a UCL file to be recognised by different programs, the type names of components and their structure must be made concrete. Such a process was undertaken to provide a configuration file for USSs.

5.4.2 System Configuration

Figure 5.10 shows how UCL is used to describe the configuration of the USS Enterprise shown in Figure 5.4. The node that has the MUM is indicated by the presence of the `MASTER` variable which is used as a flag. Likewise, one of the systems in the configuration file must be designated as the master system, similar entries would be made for the two other systems (section 5.6.2). The SID of the first system description in the configuration file is 1, the second system is allocated a SID of 2, and so on.

The `HOST` variable specifies the hostname of the node and its NID. It is necessary to describe the location of the systems/nodes in some meaningful way and the hostname's format is dependent upon the protocol used to interconnect systems. In the prototype, TCP/IP is used and the hostname is therefore given in Domain Name Server (DNS) form. The IPC mechanisms supported by the node are also listed, two of the nodes only use QNX IPC whilst the Gateway node also supports TCP/IP. Since this node is the link to the other systems it is also designated as the master.

The remaining entries correspond to the managers that run on each node. All nodes have a resource manager entry which takes a file containing its initialisation parameters. The only special manager in this system is VIS which runs on the machine with the CIG. However, one node does have the system console attached for the administrator's use.

```
USS Enterprise
{
    MASTER      ""                  // Master system

    USN Pentium
    {
        HOST    "haggis.psy", 2     // Host name and NID
        IPC     "QNX"               // Uses QNX IPC
        RM      "resnode2.ucl"      // Has a Resource Manager
        VISM    ""                  // Has a VIS Manager
    }

    USN Server
    {
        HOST    "haggis.psy", 1     // Different node
        IPC     "QNX"
        RM      "resnode1.ucl"
        CONSOLE ""                  // Has a console attached
    }

    USN Gateway
    {
        MASTER  ""                  // Master node
        HOST    "haggis.psy", 3
        IPC     "QNX", "TCPIP"
        RM      "resnode3.ucl"
    }
}
```

Figure 5.10 Example USS configuration file.

5.5 A UML Interpreter

Before examining each system component it is important to understand how the UML interpreter works because it has had considerable influence on their implementation. There are four stages to interpreting a UML description:

1. Lexical analysis.
2. Syntactical and grammatical verification.
3. Construction of the interpreter's internal data structure.
4. Semantic validation of that data structure.

The first stages were accomplished by using the `lex` and `yacc` tools (Levine *et al.*, 1992). The product of these tools was combined with a series of C++ classes to form a UML interpreter library which could be linked into any program requiring that ability. Manipulation of the interpreter is possible through the library's API.

There are two phases when building the data structure: first of all the data definition is parsed and then all instruction code is compiled into an intermediate byte-code. This section describes the general structure of this library and outlines the processes of interpretation.

5.5.1 Overall Structure

At the highest level, the structure of UML may be conceptualised as a list of universe and entity definitions. Each of these definitions may be linked to one another by inheritance or they may just be peers with a common ancestor. Every universe definition is itself a hierarchy of other components: elements, constants, properties, etc. Each entity is derived from one of the universe definitions and contains a number of scope levels with functions, variables, etc., forming yet another tree structure.

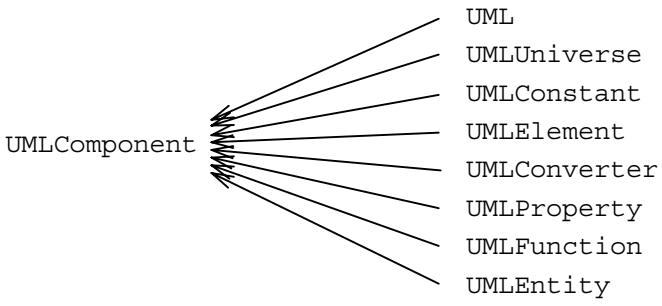


Figure 5.11 Core UML C++ class hierarchy.

Each component of UML has been implemented as a C++ class which are all derived from a common base class called `UMLComponent` (Figure 5.11). The base class holds data structures that are essential to each component class.

The `UML` object[5] acts as the top-level interface to the interpreter and the data structure representing the UML description. The other objects correspond exactly to the UML constructs described in section 4.4.

5.5.2 Interpreting the Data Definition

When a component description is encountered, its position within the data structure is first determined. At the top-level the parser may encounter any component - all but the universe and entity definitions use the dot notation. If the component is a universe then it is added to the `UML` object whilst an entity description results in its definition being added to the object. All other components require their corresponding stub declaration to be located and their description modified. Nested component definitions may be added to the relevant component data structure directly.

After all UML statements have been successfully parsed, the data structure undergoes a validation process. Universes may be derived from other universes and elements from other elements. If a component is derived from another, then that parent component is sought for and a link is made between the two components. An entity description is always derived from a universe and a similar link is made between the entity and the host universe. Failure to

[5] Instancing the `UML` class creates the interpreter and therefore there is only one `UML` object per process.

locate a parent component is a fatal error and parsing ceases. When an element is specified as the type of a property then a similar search is made and a link established.

The search for a given component starts in the current scope and, if it is not found, progresses outwards. If the host universe/element has a parent then this is also thoroughly searched and its ancestors, if necessary, until a result is obtained. Failure to locate the host component results in an interpreter error.

The way that the data structure is modified is affected by the current mode of operation, i.e. insert, replace or delete (section 4.4.3.1.9). By using these mode directives as stream modifiers it is possible to modify the UML definition in the course of usual interpretation rather than through the library API. At the completion of the interpretation, a single unified data structure has been built which holds all the UML descriptions passed to the interpreter, regardless of original physical location.

5.5.3 Instancing

At this stage no space has actually been allocated for any data. First an *instance* of the relevant portion of the data structure must be created. This could be the whole structure, e.g. instancing a universe, or just one element or built-in type, e.g. instancing a property.

When a compiler, e.g. C++, builds a map of any given data structure, each component is allocated a chunk of memory contiguous to the previous allocation. Storing all instance data together in such a container is a sensible thing to do since the data structure is static and will not change at run-time. The same technique is used in many interpreters for the same reason. However, this technique will not work with UML since the structure is dynamic and may be altered at any time.

One possible solution would be to use the same contiguous allocation of memory but store pointers to the relevant chunks in the UML data structure. In other words, each component would know whereabouts its instance data is in the container. When a change is made, e.g. a new component added, then a new container would be allocated and the existing components' data copied into it, inserting the new data in the process. A complementary technique could be used for deletion. Obviously this solution would require an amount of container memory greater in size (for the insertion case) than the existing instance data to be allocated before the process could commence. If a complex component was being altered then this could potentially be very large and at the very least result in a considerable amount of time spent copying data from one container to another.

A better approach would be to scrap the idea of storing all instance data in one place and instead store it individually. Whilst this requires a larger overhead in both memory and processing time to locate the instance data, it does mean that modifications to the UML structure do not require large memory allocations or copying. All instance data is kept associated with their definition as indicated in Figure 5.12. In this example there is one instance of the `Outer` element and two of `Inner`, one for the `innerInst` property and the other for the `local` function variable. Whilst the property instance will exist as long as that property is part of the universe definition, the variable instance will be created when the function is entered and destroyed when it has completed.

```
UNIVERSE Simple
{
      ELEMENT Outer
      {
            ELEMENT Inner
            {
                  PROPERTY number : INTEGER;
            }
            PROPERTY innerInst : Inner;
      }
      PROPERTY outerInst : Outer;

      FUNCTION Access
      {
            VAR   local : Outer.Inner;

            local.number = 1;
            outerInst.innerInst.number = 2;
      }
}
```

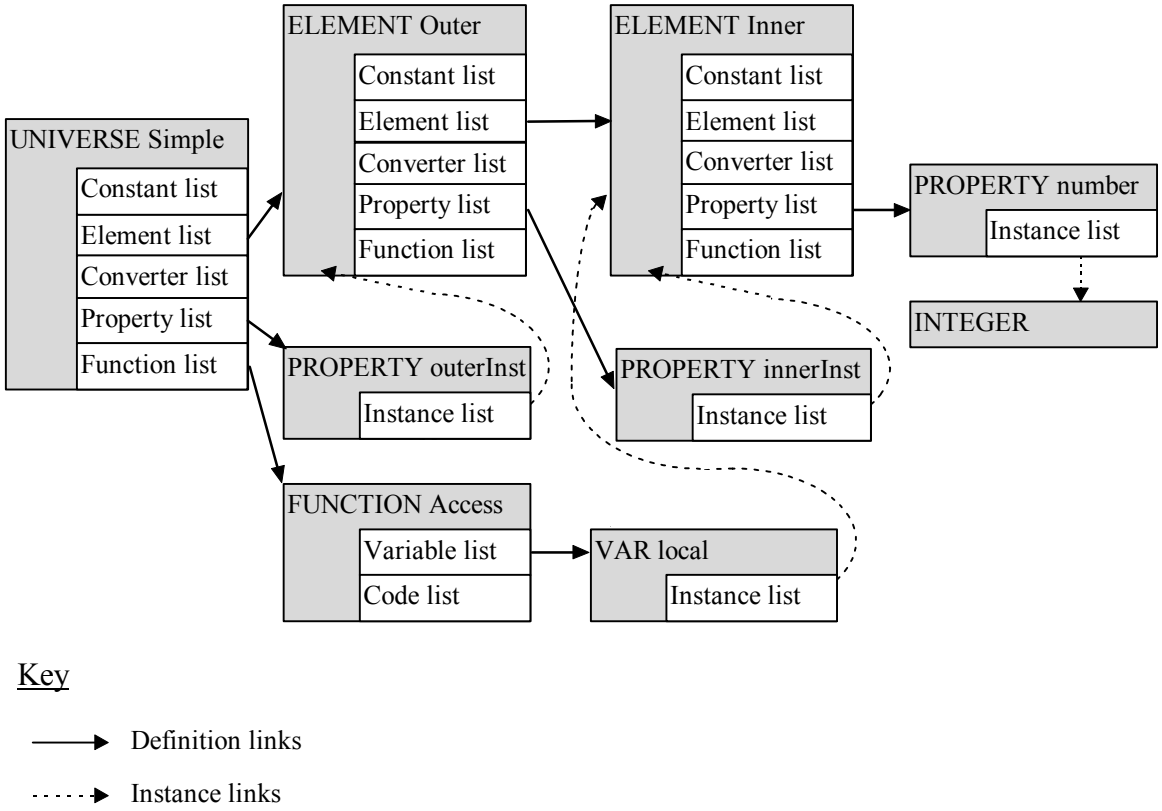


Figure 5.12 UML code fragment and the internal data structure used to represent it.

The process of instancing may be directly applied to a universe, property or function variable. In fact, for all intents and purposes, a variable and a property are functionally equivalent. Instancing a universe actually results in each of the universe's properties being instanced. If the universe has no properties then it has no state. Each property has an *instance list* which maintains a record of each instance of that property and they are distinguished through the use of an *instance identifier* (IID). An IID is a signed 32 bit integer, thus supporting 2147483648

instances during the life time of the universe[6]. IIDs are allocated to each component in the order in which they are instanced. If the property's type is an element then that element's properties are also instanced and so on until the bottom of the component tree is reached. For example, `outerInst` would have an IID of 1, `innerInst` would be 2, `number` would be 3 and `local` is 4. When a list is instanced each entry is assigned a unique IID.

```
ELEMENT Outer
{
     ELEMENT Inner
     {
          PROPERTY number : INTEGER;
     }
     PROPERTY vector    : REAL[3];
     PROPERTY innerInst : Inner;
}
```

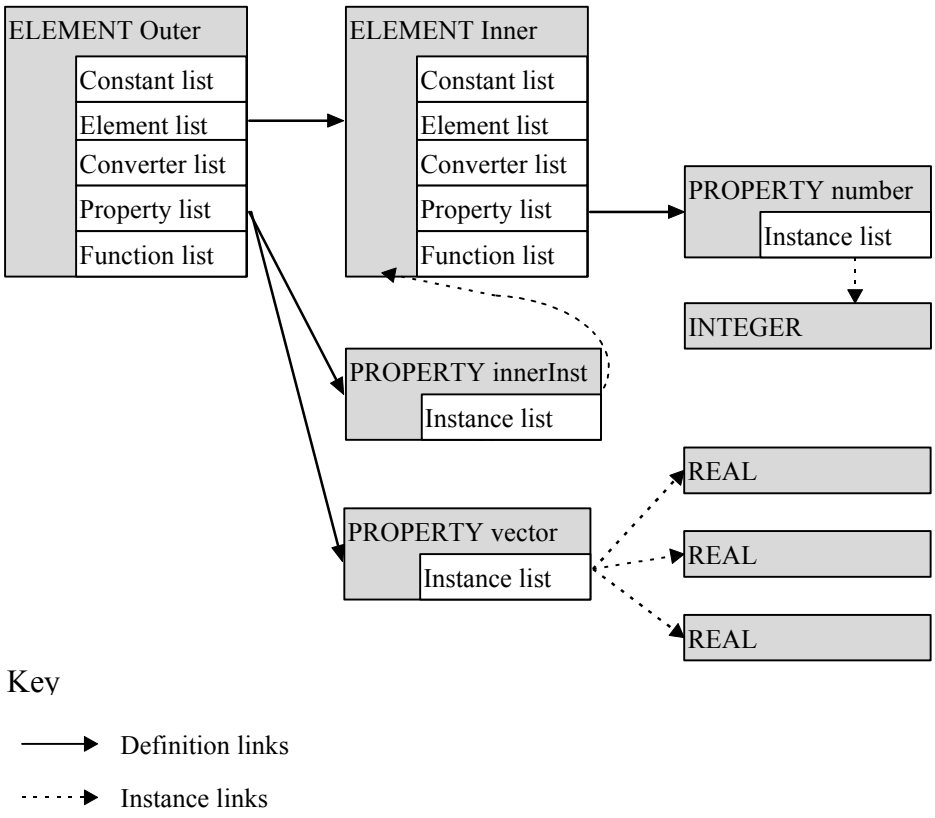


Figure 5.13 Insertion of `vector` property into element.

Consider the case when the definition is altered by the insertion of a new property - `vector` - as shown in Figure 5.13. After the data structure has been modified and validated, instancing merely requires allocating IIDs and memory for 3 real numbers and adding links to them in the instance list. The rest of the data structure has not been modified in any way and the original contents of the instance data for `Outer` have been preserved. Similarly, if `innerInst` was deleted then `vector` would be unaffected.

[6] Negative values are used for internal purposes.

In the absence of an initialiser for any given property, the default values assigned are: zero for real and integers, false for booleans, and strings are empty. This assignment is also repeated within any element that a property may instance.

State indexing (section 4.4.3.2.5) was not implemented but would require adding an extra dimension to the instance list of each property that used the feature.

5.5.4 Component Dependencies

A key feature of UML is the ability to establish a dependency on a particular part of the definition (section 4.5.4.6). The functionality to handle dependencies is defined in the class from which all components are derived - `UMLComponent`. Figure 5.14 shows its structure and that of a skeleton dependency. Just as each of the UML components are derived from `UMLComponent`, so each application uses `UMLDependency` as a basis for the information it needs to store for each dependency. An example of this specialisation is given in section 5.6.3.1 which describes how the UM uses this data structure.

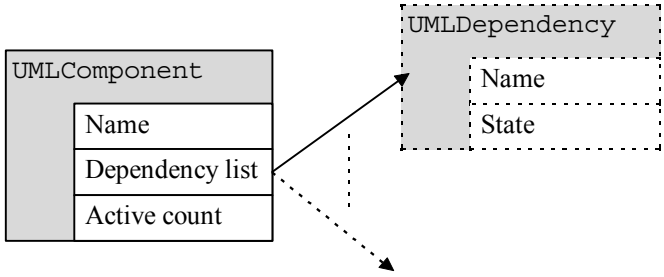


Figure 5.14 UMLComponent structure with dependency.

Dependencies are made on different components with respect to the dependent's needs. This mechanism is used internally to detect when functions which access a given component may need to be re-interpreted. It is also used by managers to keep track of changes in the values of properties, among other things.

Each dependency may be given the state of *active* (default) or *inactive*. A monitor may deactivate a dependency to avoid the overhead of removing it and then re-establishing it later on. A count of the active dependencies is maintained in the component. After a new dependency has been added or an old one removed, the monitor typically builds a dependency list. This list is usually used to process each interest in turn and perform some (often recursive) operation. If we had already registered interest in `innerInst` and now we became interested in `outerInst`, it could, at least, result in a duplication of effort and at worst, end in processing `innerInst` twice. There are therefore two ways of building a dependency list. A full list includes all components with dependencies, whereas a partial list does not include any component which is inherited from another component in the UML hierarchy with an active dependency (i.e. below an active dependency).

5.5.5 Interpreting Instruction Code

The part of the interpreter that deals with instruction code was given a low implementation priority due to time constraints. The author felt that the exact features of the programming language should be carefully considered. Also, further exploration of existing byte-code engines would be required to derive a sufficiently efficient interpreter. Furthermore, implementation was not necessary to prove the viability of the system architecture. Consequently the instruction code interpreter has not been implemented. However, some of the implementation issues are presented here for consideration by the reader.

There are two common methods for interpreting code. The first performs syntax and grammatical analysis each time, effectively interpreting the ASCII statements in their raw form. The second compiles those same statements into an intermediate code which is then executed by an automata. The overhead of parsing the original statements at execution time is large in relation to the execution of a set of pre-compiled instructions. It is true that less memory is required for the storage of intermediate code than the original ASCII text, but this must also be kept in some form if future re-interpretation becomes necessary.

For these reasons UML instruction code is first compiled into an intermediate byte-code which is stored in the data structure and may be executed by a byte-code engine at any time. During the compilation various components will be referenced, either in variable declarations, i.e. elements, or expressions modifying state, e.g. properties. If these components do not exist or there are any syntactical or grammatical faults then an error is flagged. Accesses to instance data refer directly to the data itself and therefore do not require any data structure traversal. References to properties are translated so that the instance data is accessed directly and therefore data structure traversal is not necessary. This means that any additions to the element will not require the code to be automatically re-interpreted. Deletions, however, can cause havoc.

The removal of an element or any component within an element that is depended on by code requires the re-interpretation of that code. How the functionality of the code has been affected by the change in structure cannot be ascertained without some form of artificial intelligence. Even then, comprehension of how this code segment fits into the larger picture is far more complex and would require human intervention. Consider the expression `a = b * c`. If component `b` is deleted from the definition we are left with `a = c`. This may still be valid or it may be wrong, only within the context of the rest of the code can a decision be made. Faced with the possibility of receiving dozens - if not hundreds - of requests for help from the UML interpreter, it seems sensible to at least provide some tool to aid the process. The best that can be offered is an arbitrary component expression eliminator that would remove references to the deleted component(s) whilst still retaining syntactic and grammatical correctness. The resultant code could be offered to the modifier as a potential solution and then rejected/accepted as required.

The code may, of course, be changed at any time through the API. The origin of these changes may be from a human or another program within the system. Thankfully this is a straight-forward task to complete since it is identical to the process undertaken when parsing the original code as detailed above.

5.5.6 Interpreter Embedding

As development of the simulation progresses, some definitions and associated code will be reused over and over again. The `Read` and `Write` routines declared in section 4.4.3.1.3 for managing visual information could potentially be used in every entity. Translation of such UML code into the native IL would be sensible for performance reasons. Access to the interpreter's data structures is possible via the library API and the execution of native machine code (rather than UML) will be transparent to the application. The IL routines are usually placed in a library and linked in with each application that needs them. The ability for an entity to migrate to other nodes need not be affected if:

1. The destination node has its own native version of these routines.

2. The original UML code is at hand and may be used when native code is not available.

Permitting the interpreter and ILs to interact provides a powerful basis with which simulations may be developed. UML code may be used for lightweight tasks and rapid prototyping of more complex functions which, when finalised, may be coded in the IL.

5.5.7 Persistence

Since the complete definition is either represented by a data structure (in the case of the data definition) or by the original text (in the case of the instruction code), it is possible to output any part of a UML definition at any time. This ability is very useful when changes have been made at run-time and the original definition is now incorrect.

To migrate an entity requires the transfer of its essence from one place, i.e. the UML definition *and* its current state. Fortunately the definition can always be reconstructed from the state so it is only necessary to send the latter. The same process is also required in order to save the current state of an entity to backing storage so that it may be reloaded in the future.

The state is the sum of all the instance data and packaging it, by necessity, involves the manipulation of binary data. If this package will be sent to another node then, in a heterogeneous network, it may not share the same architecture. Following the decision made in section 5.3.4, three routines are defined in the IL for every component: `size`, `pack` and `unpack`. The `size` routine traverses the given definition and estimates the size of each of its components, producing a grand total at the end. This figure is used to allocate a buffer into which `pack` stores the data by once again traversing the data structure. Each component's instance data is preceded by a small header providing vital information to aid its extraction by `unpack`. When packing or unpacking the data on a little-endian machine no binary conversion is necessary, overheads are only incurred on big-endian systems.

5.6 Universe Manager

There are three main stages to the execution of the UM. First of all the UM's node must be initialised, at which point it is ready to join the network of other nodes comprising the system.

Once this connection has been established, it enters an event loop which processes service requests that are sent to it and also generated internally.

5.6.1 Node Initialisation

As the first process to start, the UM is responsible for configuring its node and if it is the MUM, organise the system. After the PML has paused its initialisation, the first action taken is to process the configuration file. Its local node and the master node information is located, as well as location information for the other systems if it is the MUM. As each node/system is processed the UM builds a routing table for those systems that are connected via TCP/IP. Now that the SID, NID and PID are known, the PML completes the initialisation of the IPC mechanisms.

At this point the execution paths differ for SUMs and MUMs. If it is running on the master node then the location of the UML definition[7] is verified and interpreted. All SUMs locate their MUM and send it an `ACTIVATE_UM` message. Afterwards, all UMs create any managers that are configured for their node, starting with the RM and then the specialised managers. The creation of a console is initiated by the administrator and may be performed at any time.

5.6.2 System Initialisation

After manager creation, system initialisation is completed. The MUM waits for activation messages from each SUM which it acknowledges. This acknowledgement changes the node's state to *alive*. When all nodes are alive the system itself is deemed to be alive.

In the prototype a multi-level hierarchical system organisation is not supported, rather a simple master/slave structure has been adopted. In the same way that there is one master node in a system, there is one master system (MUSS) and zero or more slave systems. Any communications that must be sent to other systems are sent directly to the MUSS which routes them to all the other systems. Therefore, after the MUM has initialised its system, the address of the MUSS is sought and stored explicitly for future use.

5.6.2.1 Load Balancing

Rather than obtain a full RP from each RM, the prototype uses a simple CPU rating in the current load-balancing algorithm to determine on which node the declared entities in the universe definition should execute. Each time an entity is created, the optimum distribution of processes between nodes is recalculated and the entity is allocated to the node that has the largest difference to its optimum load. Table 5.5 shows the debugging output from the load-balancing algorithm. The figures inside brackets represent the ideal number of entities for each node if another entity is created, whilst those outside are the current distribution of entities.

---

[7] The filename is passed as a command line parameter.

Current Entities	Entities on Server	Entities on Gateway	Entities on Pentium
1	0 (0.204)	0 (0.224)	1 (0.572)
2	0 (0.408)	1 (0.447)	1 (1.145)
3	1 (0.612)	1 (0.671)	1 (1.718)
4	1 (0.816)	1 (0.894)	2 (2.290)
5	1 (1.020)	1 (1.118)	3 (2.863)
6	1 (1.224)	1 (1.341)	4 (3.435)
7	1 (1.427)	2 (1.565)	4 (4.008)
8	2 (1.631)	2 (1.788)	4 (4.580)
9	2 (1.835)	2 (2.012)	5 (5.153)
10	2 (2.040)	2 (2.235)	6 (5.725)
11	2 (2.243)	3 (2.459)	6 (6.298)

Loading was based on CPU ratings of 260, 285 & 730 respectively. Figures in brackets represent the new optimum load for each node to 3 sig. fig.

Table 5.5 Sample entity distribution over three nodes.

The first row of the table shows that the first entity was allocated to Pentium. The fastest and least loaded node is always chosen for the target when the next entity is created, which in this case means Gateway with a predicted loading of 0.224. This result is confirmed by the second row in the table which also shows that the next entity will be allocated to Server and so on. When there are ten entities the home for the new entity is Gateway. This is because Pentium is overloaded by 0.275, Server can only handle 0.04 more entities and Gateway has room for 0.235 entities. A total of the entities active on each node is kept at all times.

Currently there is no way of associating an RP with a specific entity so each entity is allocated an initial default profile.

5.6.2.2 Entity Creation

Originally it was planned for the MUM to extract the relevant portion of the UML definition and send it to the destination node's UM. However, if an entity should migrate to a node that does not have the entity's definition it must be sent prior to the migration, thus increasing the time taken to complete this operation. Therefore each UM has a complete copy of the UML definition. Since the instruction code part of UML has not been implemented, the entity's functionality is written in the implementation language and executed in place of interpreted code (section 5.8.1). Normally there would be one generic entity process with a built-in UML interpreter to start, but because functionality may differ between entities, a specific executable must be identified. The prototype takes the name of the entity and translates this into the name of an executable that exists within the search path of each UM[8]. A `CREATE_ENT` message is then sent by the MUM to the target node indicating the name of the executable. On receipt of this message the process is started, indication of success is sent back in a

[8] This path can be modified using the `ENTPATH` variable in the node's configuration section.

CREATE_ENT_ACK message and entity execution continues as usual (section 5.8). Of course, if the entity is executed locally then the process is merely started and the MUM moves onto the next entity.

5.6.3 Managing Processes

Information about each process running on the node is held by the local UM in a *process list*. The structure of a process entry is shown in Figure 5.15. Every process is allocated one of six types: RM, ENT, MAN (special manager), MUM, SUM and CON (console). There are three states that processes progress through during their lifetime. After execution has started, but before the process has been allocated a UPID, it is allocated the state of *genesis*. When the initial handshaking is over and the process is ready to satisfy service requests it is said to be *alive*. During the termination process, after it has ceased to function in the simulation *per se*, the process is said to be *dead*. When termination is complete the entry and its dependent structures are removed from the list.

Any given UM holds information about every entity and manager running on its node; if it is the MUM, information on any SUMs is also held; if it is a SUM, its MUM's details are stored. Treating parent and child UMs as processes running on its node simplifies certain procedures that the UM must perform, e.g. dependency management (described below).

5.6.3.1 Component Monitoring

When a manager wishes to monitor a given UML component, its absolute name (using dot notation) is sent within a UML_MONITOR message. After verifying that this component actually exists the manager's information is found within the process list, a new dependency is created and added to the process' *dependency pool*. The pool is essentially a fixed size array which provides fast entry lookup. As dependencies are removed, gaps appear but these are filled as new dependencies are added.

The UMDependency information is derived from UMLDependency as described in section 0 and adds a pointer back to the owner's process entry (Figure 5.15). This organisation permits any *process* to locate all of the components it is dependent on and any *component* to determine which processes are dependent on it. Although the framework is here to support dependencies on any component, only monitoring of properties is currently implemented.

The monitor ID returned to the manager is actually the component's index in the process' dependency pool. The UM must now inform all relevant processes that a new dependency has been established using a UML_MONITOR_ACK message. As each entity is processed a new dependency is also added to their pool; its index provides the monitor ID to be used in communications with this entity. Both the MUM and the SUMs are also informed using the original message sent by the manager. Each add a dependency to the sending UM's process entry and inform the sender of the monitor ID to be used in further transactions regarding this component. Without keeping a process entry for parent/child UMs, this procedure would be far more complex than necessary. If an entity is created after all dependencies have been established then a current list is sent as a stream of separate messages.

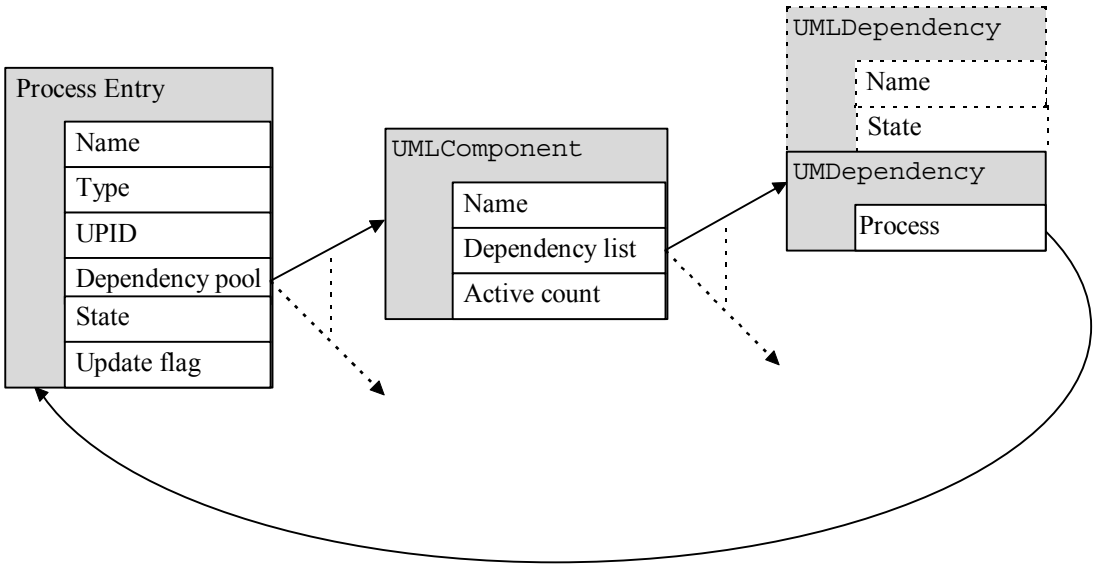


Figure 5.15 Structure of the information held for each process.

5.6.3.2 Component Updates

When an entity sends a state update to the UM, its process information is retrieved and the dependency pool entry described by the monitor ID in the message is extracted. From this point a list of those processes dependent on this state is available. Each dependent's unique monitor ID is extracted from their pool and placed into the message before it is forwarded to it by the UM. Figure 5.16 presents an example where the component state has an ID of 1 when it is sent to the UM, but has the values of 4 and 2 when forwarded to the two interested managers.

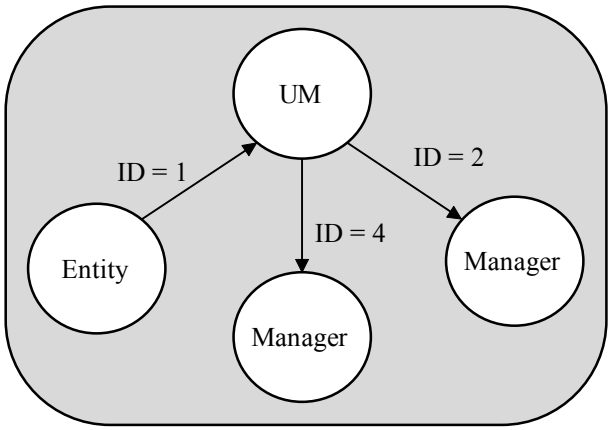


Figure 5.16 A state update uses a different monitor ID when sent to each dependent.

No extra space is required to store each monitor ID because it is the index into the dependency pool. The only computational overhead incurred is a simple pool lookup as each dependent is processed. Constraint functions were not implemented because they rely upon the UML instruction code interpreter which was also not implemented.

5.6.4 Processing Service Requests

Two features common to all component implementations are the *event loop* and the *action queue*. When an internal function wishes to perform more than one action, e.g. send a message, or can/needs to spread its work over a period of time, then it enqueues a token representing the pending action (with parameters) in the action queue. The event loop checks if there are any external service requests which it processes first to maintain responsiveness. If there is not a message waiting then it dequeues the next action and performs it. If there are not any actions to perform then the process simply blocks until a service request arrives. It is not uncommon for one action to enqueue another during its execution.

One action that must be performed in the initial stages of a UMs lifetime is waiting for all entities and managers to complete initialisation, the specifics of which are described in the following sections. When all entities and managers are alive the simulation loop is entered which sends a `UML_UPDATE_NOTIFICATION` message to each entity and manager. After all entities have updated, a `UML_UPDATE_COMPLETE` message is sent to all managers and after they have updated the next notification message is sent and so on. The state update process triggered as each entity completes its update has already been described and the following sections discuss this and the managers actions in more detail. Other service requests/actions that are intermingled with this sequence are location requests, entity executions, synchronisation requests, etc.

If an entity should terminate abnormally and a destruct message has not been issued then the UM will do so on behalf of the late entity. This ensures that the simulation does not become full of zombie entities whose state copies are still being maintained by managers.

5.6.5 Entity Migration

In order for entity migration to be implemented it is necessary to have some basis upon which to make decisions about node loading. This was done through the use of CPU consumption alone. However, without a fixed time frame to relate these measurements to, a CPU usage is useless. This fixed period would normally be provided by the scheduler and equate to one simulation step, but since a full scheduler was not implemented a simple step duration threshold was used for the migration test presented in section 6.5.4. The intention is to keep the simulation step duration below the threshold through use of migration. Each step, the RM totals the amount of CPU used by the entities and if it exceeds the threshold the migration mechanism is be triggered. In this prototype the MUM does not decide when migrations should take place but relies upon each RM to volunteer entities.

When the mechanism is invoked, the entity with the largest CPU usage is identified and its UPID sent to the MUM in a `MIGRATION_REQUEST` message. The requests, of which there may be more than one generated by different nodes each step, are enqueued and then processed at the end of the current simulation step. The source node of each request is excluded from selection in the load-balancing algorithm and the optimum distribution is calculated as if the system has one less node. Once a suitable target node has been found, an entry is added to the MUM's *migration list* which details those entities in the process of migrating and their current status; specifically, their name, source node, target node and source UPID.

The next stage is to create a copy of the entity on the target node using the normal creation procedure. Once this has been done, a `MIGRATION_STATE_REQUEST` is sent to the original entity which packs up its entire state and returns it to the MUM in a `MIGRATION_STATE` message. This is then forwarded by the MUM to the newly created entity which unpacks it and, upon success, sends a `MIGRATION_STATE_ACK` back to the MUM. Finally the original entity is terminated by sending a `DESTROY_ENT` message to the entity's UM, `MIGRATION_NOTIFICATION` messages are sent to all managers (including SUMs) and the entity's migration list entry is removed. The notification message simply contains the old and new UPIDs for the entity and enables the managers to update their internal data structures accordingly. The UM on the source node uses this information to re-route any messages that are sent by processes unaware of the migration. After forwarding the message, the UM sends the originator a migration notification message so that this does not happen again.

Currently any error that occurs during the entity migration, e.g. failure to create the target entity, is treated as fatal and the migration request is ignored.

5.6.6 System Interaction

The multi-system functionality that has been implemented is limited to group initialisation, termination and the transmission of changes in the UML definition. Inter-system user functionality has not been implemented, e.g. shadow entities, because it is hard to demonstrate in a thesis and was therefore given a low priority.

5.6.7 System Termination

A system termination is invoked from the MUM by first sending termination messages to each SUM. The MUM and SUMs then send termination messages to their local managers and destruct messages to all their entities. Once all processes on a slave node have terminated the slave informs the MUM that the node is shutting down with a `DEACTIVATE_UM` message. Finally, when all the local processes on the MUM and its slaves have terminated, the MUM ends execution.

5.7 Resource Manager

The implementation of the RM is quite simple because there is no scheduler. Subsequently the RM keeps track of the resource utilisation for its node and makes rudimentary judgements about its loading.

5.7.1 Resource Consumption

Each resource has been implemented as a class derived from one base class (Figure 5.17a). An RP is composed of these different types: a list of CPU consumption (for multiprocessor systems), a list of memory usage (used in those systems with special memory architectures) and a record of space used on different storage devices. The totals of each of these are also

stored and is supplemented by the network usage (Figure 5.17b). The prototype actually only makes use of the CPU information.

The RM maintains a resource history for each process (Figure 5.17c) which contains the process' last RP, its current profile and a prediction of future resource requirements (currently unused).

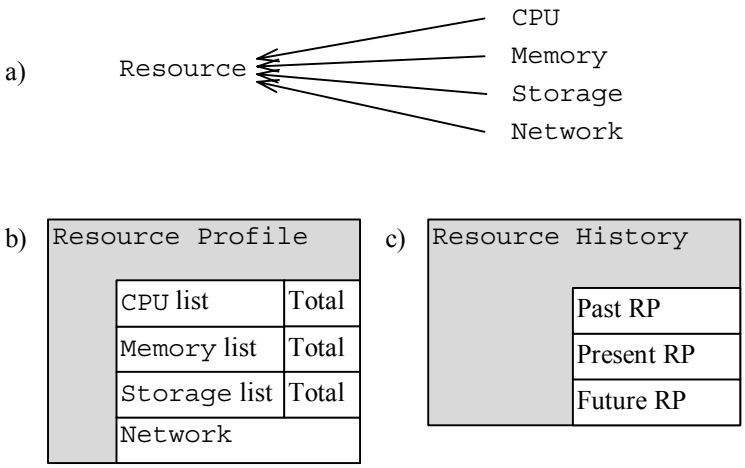


Figure 5.17 Resource consumption representation.
a) class hierarchy; b) Resource Profile structure; c) Resource History structure.

5.7.2 Initialisation

Each node's resources are detailed in a file (written in UCL) which is passed as a command-line parameter to the RM when it is started. Figure 5.18 shows an example configuration of the `Pentium` node which details the CPU type (an Intel Pentium/90), the total system memory, backing storage and network link bandwidth. For the migration tests a `CYCLE` variable was used at the top level to specify the threshold duration of the simulation step in milliseconds.

5.7.3 Services

After the configuration information has been processed, the main event loop is entered. Initial work usually consists of processing the RPs sent by each entity as it is created and keeping the UMs informed of the current loading. During the period before the system goes live it is not possible for an entity to overload a node since it has been carefully allocated by the MUM. However, as soon as the entity starts executing it may provide modifications to its RP based on its expected resource consumption. Since a full scheduler was not implemented, this detailed information was not needed. For the same reasons, the RM does not keep the MUM informed of node loading. Instead the RM tells the MUM when load balancing is necessary.

Since this prototype instills the progression of the simulation with the MUM rather than the scheduler in the RM, an `UPDATE_NOTIFICATION` message is sent to the RM at the *end* of each simulation step. This is the RM's cue for assessing CPU usage and when this is complete an `UPDATE_COMPLETE` message is sent back to the UM. The simplest

information on a process' execution time under UNIX-based operating systems is provided in the form of user and system times. These represent the total CPU used by the process when executing system calls (system) and when executing application code (user). The current RM adds these figures together to get a CPU usage figure for each process. By monitoring the previous usage the process' consumption for the last simulation step can be ascertained.

```
RM
{
    CPU Pentium_90MHz
    {
        Manufacturer    "Intel"

        Integer         0.849    // BYTEMark integer index
        FloatingPoint   0.881    // BYTEMark floating index
        ICache          8        // Kb
        DCache          16       // Kb
        IntThreshold    90.0     // %
        FPThreshold     90.0     // %
    }

    MEMORY Main
    {
        Size            24576    // Kb
        Access          70       // ns
        Threshold       80.0     // %
    }

    STORAGE Primary
    {
        Size            524288   // Kb
        Access          12       // ms
        Threshold       95.0     // %
    }

    NETWORK Ethernet
    {
        Bandwidth       6.0      // Mbps (Effective)
        Threshold       40.0     // %
    }
}
```

Figure 5.18 Example node resource configuration used by a RM.

When the migration mechanism is being used, the total of these times is used to decide whether the entity with the highest CPU usage should be migrated. Currently the CPU thresholds are not used, instead the step duration variable (`CYCLE`) is consulted for the desired time. If the total CPU time used by all entities exceeds this time then a migration request is sent to the MUM. The RM is informed of a successful migration with a `MIGRATION_NOTIFICATION` and subsequently removes the entity from its calculations.

5.8 Entity Library

The core entity functionality has been placed in a library which works on two levels. Once initialised, its event loop enables it to correctly interact with other processes in the system and, through the use of a function call-back mechanism, can be tailored for a specific purpose. The source code of an example entity can be found in Appendix B.

5.8.1 Initialisation

Following PML initialisation, the call-back table is reset and specific call-backs may be registered. An entity handles all the UML messages in addition to those dealing with RPs, location responses and monitor acknowledgements. The first message processed by the entity is its RP which can then be modified. After locating the RM, the RP is sent to it and a request is made for the entity's UML definition.

Normally entity behaviour would be exhibited through execution of UML code, but since the instruction code interpreter has not been implemented, functions written in the IL must be used. Typically the only call-back used is that for the `UML_INIT_DEF` message which is used to send the entity its definition. At this point the entity's UML `Construct`, `Update` and `Destruct` function declarations are located and defined as embedded IL routines as opposed to UML code. When these functions are executed by the UML interpreter, the IL routine is called. Access to the state information is obtained through the UML API.

5.8.2 Service Requests

The first external events received by the entity are indications of monitored components in the form of `UML_MONITOR_ACK` messages. Unlike the UM, the only information that the entity need keep track of for each dependency is the monitor ID contained in the message. This dependency list is rebuilt each time a new monitor notification is received.

Upon receipt of a construct message the UML interpreter is instructed to construct the entity's state. On completion an instance ID is returned which is used in all further accesses to the state information. The `Construct` function is then executed, thus initialising the state and is followed by the enqueuing of the action to send initial state updates to the UM. Receipt of an update results in the same execution-action sequence. The current component dependency list is used to determine which state updates to send. Asides from executing the destruct function, no further action is taken when an entity destructs. The PML, by default, informs the UM of the process termination and whether it did so naturally or not.

When a `MIGRATION_STATE_REQ` message is received by the entity, it packages up its complete state and sends it back to the UM in a `MIGRATION_STATE` message. Upon termination the entity destructs as normal. When the target node is sent the state message it instances its definition and unpacks the state into the newly created instance. The construct function is not called and a `UML_CONSTRUCT` message is not sent to the UM. From this point on, however, the target entity takes over all processing from the original and operates normally, issuing state updates as necessary.

5.9 Manager Library

The manager functionality has been structured in a similar manner to that of an entity. On its own, the library will interact correctly with the other process' in the system but does not perform any special manager-specific tasks. This higher-level functionality is added through the call-back mechanism. Appendix B contains an example of this library's use.

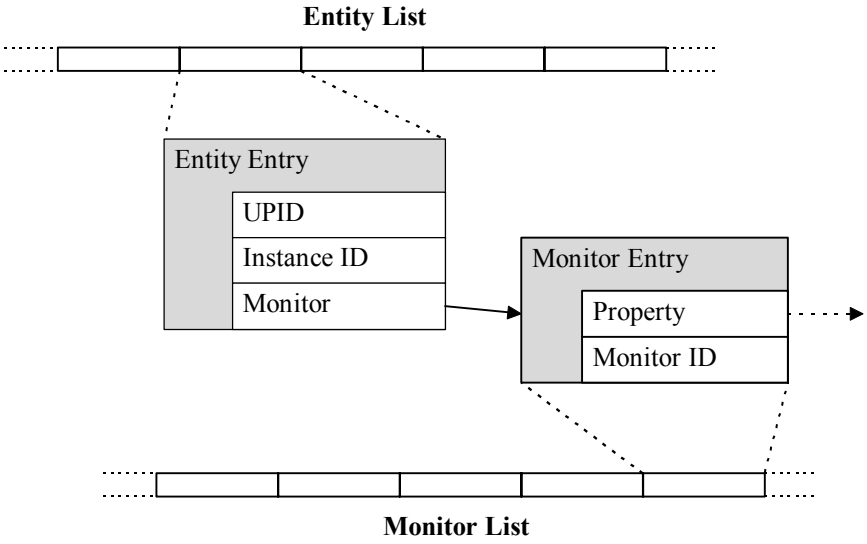


Figure 5.19 Structures used to keep track of entities and their component dependencies.

5.9.1 Initialisation

Following the usual process initialisation, the manager is sent the complete UML definition and (through a call-back) registers interest in the specific components it uses. Each manager maintains a *monitor list* with an entry for each component it is monitoring (Figure 5.19). An entry consists of a pointer to the relevant portion of the UML data structure for that component and the monitor ID used in communications with the UM. The three other essential call-backs are those for `UML_CONSTRUCT`, `UML_UPDATE` and `UML_DESTRUCT`. It is within these functions that the heart of the special manager's functionality is embodied. An example of their use is given in section 5.10.

At the lowest level the manager keeps an *entity list*. An entry is added to this list on receipt of a construct message, modified by an update message and removed when an entity destructs. An entry exists for each monitored component held by each entity. When a construct message is received for a component, that part of the UML data structure is instanced and the contents of the message unpacked into the state instance. The instance ID is stored in the entity list entry along with the entity's UPID and a pointer to the relevant entry in the monitor list. This enables the location of all state information related to a specific entity with minimal redundancy.

5.9.2 Simulation Loop

Each simulation step starts with the reception of a `UML_UPDATE_NOTIFY` which can be used via a call-back to perform preliminary work for each update. When an update message is received the monitor entry is located using the message's monitor ID. Then the component's state is located by searching the entity list using the entity's UPID and the monitor entry as keys. The new state is then unpacked into the instance and the update call-back executed if present. When a `UML_UPDATE_COMPLETE` is sent by the UM the simulation step has

concluded and the manager may perform (via call-back) any final calculations before the next step. The return of a status message to the UM indicates that the manager has completed its work. This start/stop message system is necessary because an entity will not send an update unless that component has been modified. Therefore there is no way for a manager to determine whether all updates it should be sent, have been sent. A destruct message results in the deletion of that entity's component instance and then the removal of the relevant entry from the entity list.

When a `MIGRATION_NOTIFICATION` is received, the manager locates the old entity's entry in the entity list and replaces the UPID stored therein with the new address in the message. No other action is needed.

5.10 Visual Manager

The prototype VIS implementation does not interface to a CIG since it was not deemed necessary in order to demonstrate the effectiveness of the USS architecture. In fact, it is not used when evaluating the system's performance in the next chapter, but it is presented here as an example of a special manager implementation.

The code used to explore the viability of real-time VE displays was available for use (section 5.1.1) but was not utilised for two reasons. Firstly, there is no way to satisfactorily demonstrate such a feature in a thesis. Secondly, graphics and API speed is totally CIG dependent and would only confuse any analysis of the manager's performance. Therefore, everything apart from the actual calls to the CIG's API was implemented.

5.10.1 Initialisation

Following the standard manager initialisation the prototype VIS registers interest in the `Base.models.visual` and `Base.models.position` properties (section 5.10). At this point the CIG would also be initialised and initial parameters set, e.g. viewpoint position, etc.

VIS registers call-backs for all construct, update, destruct and update-complete messages. As each entity constructs, VIS receives a stream of construct messages which are acted upon by the call-back function. This is responsible for creating the initial visual representation of the entity in the CIG database.

5.10.2 Simulation Loop

As updates are sent to VIS, the update call-back is executed which is used to move the entity's representation and if necessary, modify it. On receipt of an update complete notification the new scene is rendered and the manager has finished its work for the current step. Destruct messages result in the removal of the representation from the CIG database.

5.10.3 Entity Enhancement

The extra functionality needed by any entity wishing to manipulate its visual representation is provided in the form of a library. Whereas this could be provided as importable UML code, it is currently IL code which is linked into the ENT executable. The `Read` and `Write` function definitions (section 4.4.3.1.3**)** are supplemented with internal routines which may be used to manipulate the `Visual` element data structure.

Therefore, an entity's construct call-back function will build the visual representation, either from file or by code. The update call-back modifies the state as necessary and the destruct call-back closes the library.

5.10.4 VIS Summary

The current VIS implementation is very basic but it performs the essential operations required of it. Since all the complex operations are hidden in the manager library, the developer can concentrate on what the manager should be doing and implement it with the minimal coding.

5.11 Console

The console implementation is a hybrid of a manager and an entity in that it receives most messages in order that it may keep track of the system's status. A command-line interface provides the opportunity to display this information and issue simple commands. An entity creation, destruction or migration request may be sent to the UM from the console, as can UML code. The console keeps an up-to-date copy of the complete universe definition although it does not maintain any instance data. The current functionality is quite limited and was used for testing purposes only.

5.12 Further Improvements

At this stage, it is apparent that a number of enhancements can be made to the prototype.

5.12.1 Configuration

The configuration information required by child processes, e.g. the RM, is currently passed to them as a filename in their execution parameters. This has two disadvantages: firstly, it introduces a dependency on backing storage and, secondly, it increases the process initialisation time. If this information was passed to them by the UM, both these problems could be overcome. This would not require changing the current configuration file format and could be sent in its native ASCII format.

5.12.2 Multi-part Messages

Presently the PML relies on the operating system to break large messages into smaller packets for transmission. This ability is not supported by all IPC mechanisms and therefore the

addition of PML controlled multi-part messages would be advantageous. This would also reduce the amount of buffer space required to send a message and permit the construction of messages whose total length is not known when the first part is sent.

5.12.3 State Encoding

With the ability to gradually build a message, the estimation of state size prior to encoding may be removed. Instead the state may be encoded directly into a multi-part message thus substantially reducing the time taken to send state updates.

Alternatively, memory could be allocated during packing, building a linked list which is then traversed when copying the state into the fixed size message buffer. This, at least, removes the need to estimate size initially.

5.12.4 Persistence

The current implementation assumes that a simulation will run to completion before the system terminates. Therefore no provision is made for state persistence such that a simulation may be saved and reloaded at a later date. In order to realise this, an entity could be sent a `TERMINATE` message before destruction which would be its queue to save its state to backing storage. Upon restarting a simulation the entities would be created as before (but possibly not on the same node) and during construction their state loaded from backing storage. Managers can rebuild their internal data structures from the events that would take place upon restarting the simulation, e.g. entity creation, initial state transmissions, etc. It may be necessary, however, that those structures unique to each manager are also saved for use when the manager re-initialises.

5.12.5 Message Elimination

The three messages `UML_CONSTRUCT`, `UML_UPDATE` and `UML_DESTRUCT` sent to an entity should be replaced by a UML message which simply executes the `Construct`, `Update` or `Destruct` function respectively. The resulting state updates generated by these calls would be returned in a standard *state* message which would include the name of the function that generated the data. All such remote code executions would operate in the same manner. The current shortcut was taken because the UML interpreter was not complete.

5.12.6 Entity Synchronisation

Synchronising an entity involves the transmission of multiple messages detailing individual monitor notifications. In this special case it would be preferable to send a single message containing all notifications, thus reducing the UM's overhead for this operation.

5.12.7 Function Access

At present, anybody may execute a function in an entity if it knows its name. This could be changed by providing a function hiding mechanism, e.g. a `PRIVATE` keyword to be used in the function declaration (not definition). Any attempt by a remote process to execute a private function would result in an appropriate exception generated by the interpreter.

This technique could be generalised by ensuring that any private function cannot be executed outside its scope. In Figure 5.20, `unprotected` may call `protected` since it is in the same scope but `control` may only call `Inner.unprotected`.

```
ELEMENT Outer
{
    ELEMENT Inner
    {
        FUNCTION unprotected;
        FUNCTION protected PRIVATE;
    }

    FUNCTION control;  // Can't access protected
}
```

Figure 5.20 Example use of the `PRIVATE` keyword to reduce function access through scope.

5.13 Summary

Before the details of the prototype USS were given, the implementation of a simple worst-case scheduler was described which has been used to enforce a constant-rate display. The experience gained by the author during this implementation and its subsequent use indicated that implementing scheduler functionality at the application level was not practical. The USS prototype implementation presented therefore did not make use of the scheduling aspects detailed in the design.

A layer of abstraction is introduced in the form of the PML in order to shield the USS processes from each operating system's idiosyncrasies. Presently it is only used to provide a messaging service between both local and remote processes. The simple configuration language was then described and a typical example of its use presented in the form of the USS configuration file. The structure of the UML interpreter was described in terms of the data definition and instruction code sections. This included a detailed explanation of the complex data structure used to hold the model description and its instance data.

Each of the required system processes were dealt with in turn, describing the implementation of the basic operations they perform and services they provide. Special attention was given to the important data structures and how they are utilised at run-time. Most of the UM's functionality was implemented including an elementary migration and load-balancing mechanism (using a minimal RM). The bi-directional data structure used by the UM permits the location of all components that a given process is dependent upon and *vice versa*. The operations involving state transmissions and monitor IDs were described in conjunction with details of the relevant parts of the manager and entity implementations. The core entity and

special manager functionality is provided as libraries which are specialised through the use of UML code and call-backs. An example of this is given with reference to the Visual Manager.

The chapter concluded with a few improvements that may be made to the current implementation. These functional changes will be supplemented by performance enhancing suggestions in the next chapter.