Chapter 4 Gait Generation in Hexapod Robots and Local Modeling Techniques

4.1 Introduction

In the preceding Chapter the design and construction of an autonomous hexapod robot called ELMA, was detailed. This robot uses the *LonWorks*® distributed control system to connect a number of leg controllers together along with a head, or sensing, node. The simple Central Pattern Generator (CPG), briefly explained, demonstrated that the robot was stable and had sufficient battery life to function for a useable period of time before recharging became necessary. Furthermore, the tests using a CPG demonstrate clearly that a low cost fieldbus can be used for co-ordinated applications such as walking robots. It can be inferred from the experimental results on bandwidth usage that the network could support up to three times the number of controllers currently present with minimal degradation in performance. These extra controllers do not have to be leg nodes, for instance WALTER incorporates an infrared remote control tail and this could be used within ELMA (WENN94). The addition of these devices would reduce battery life. However, on an architectural level, the addition of devices using this distributed control system is simplified.

This Chapter goes on to explore several software architectures that have been tested on the robot. As such, it forms an extended case study on the implementation of distributed control schemes on mobile robots when applied to networks of simple processors. Whilst the robot is quite sophisticated for its size and weight it is lacking some sensors that would provide useful feedback to the control algorithms. In particular joint angle sensors on the legs are missing since the servo units operate in an open loop fashion and do not provide position information. One way round this is to connect directly into the servomotors' positioning circuitry as has been done with Genghis, ANGL89, which uses the same servomechanisms as ELMA. This has not, as yet, been attempted on ELMA and so a different approach has been used so that development and verification of the control algorithms could continue. This approach entails the development of functional blocks that reproduce missing dynamic behaviour. These blocks are inserted immediately before the joint controllers in the various architectures. They also are shown to form part of the performance model of a device that, as mentioned in Chapter 2, goes to make up the full Local Object Model for the leg.

Once an algorithm has been developed for the robot, it then becomes necessary to monitor its behaviour. This entails examining many values within the robot over a period of time. Initial tests with ELMA were carried out using a custom written development tool that could represent a specific subset of network parameters. Whilst this was useful for this limited application substantial changes in network architecture required modifications to the program and demonstrated the need for a more flexible visualisation tool. The limited number of nodes on ELMA forms a system that is easy to visualise however with complex installations the task of observing and interacting the whole system becomes difficult. In large plants the wealth of information can be difficult to visualise and much time is spent developing graphical representations of the system that ease data assimilation for the user. Referring back to the LOM concept a compact extension to this model is shown that allows graphical encapsulation of the device state. Additionally the remaining models in the Local Object Model are explored briefly.

There is a wide variety of different mobile robot controllers. The existing range can be divided into two main types, namely those that are completely pre-programmed and those that in some way learn behaviours, KELL97, ARKI98. Turning attention to the first case, there is a range of different types within this classification ranging from *hard-wired* controllers that operate without regard to external influences (TODD85), to simple reactive systems that offer a deterministic response to a given machine state and input domain, BROO86, MATA92b, FERR94. In the second learning case considerable development is underway coinciding with interest in Artificial Intelligence (AI); the interested reader is directed to material such as LUGE89 for a general introduction to AI. It is not the intention of this thesis to propose radically new structures for the control of autonomous walking robots using either of these methods. Instead, some existing control architectures have been used to verify the behaviour of the fieldbus network, in conjunction with a novel method for end effector (or ‘feet’) trajectory planning that does not rely on pre-programmed or reactive behaviour. In addition, Section 4.3 and Section 4.4, investigate the use of behavioural blocks to mimic incomplete hardware information (due to a lack of sensors) and at this point, the Performance Model of the LOM is introduced.

4.2 Central Pattern Generators

The most fundamental level software used by ELMA is based on a Central Pattern Generator (CPG). In fact the earliest walking gait control programs were also based upon CPGs since computational power was limited and there was more interest in verifying the hexapod structure rather than advanced behavioural control models, MCGH66, SONG89. The CPG can be likened to the specialised animal neural structures that produce sequencing information for the legs as described by Lewis et al. in LEWI93. There is still conjecture as to where precisely simple animals (such as cockroaches) produce the firing signals for their motor neurones with some researchers advocating central brain based generators and others moving the control generation to the segment close to the leg, HILL67. Robot hexapods have successfully been used to test the various hypotheses in a machine setting. For example, Chiel et al. (CHIE92) has successfully demonstrated hexapod control using both distributed and centralised pattern generation, albeit in the overall context of a reactive controller. It is worthwhile to note that the investigation into insect locomotion has so often been carried out on robots since they provide a real world problem set which is not completely trivial yet not too complicated to prevent rapid development, LEWI93.

Within a distributed control system such as a fieldbus there are a number of architectures that can be implemented that use a non-reactive central sequencer for leg positioning and gait control, these are covered in the following sections.

4.2.1 Centralised Leg Positioning

In this architecture, a *brain* node has a table of leg positions. Each entry in the table corresponds to the desired position of each leg within the overall sequence of the gait. As the brain indexes through the table, at each point the new values are output via the network to the leg nodes. In this way the leg nodes perform no localised processing or scheduling other than signal conditioning and translation of the arriving signals. This extremely simple arrangement with each leg effectively operating in open loop is useful for verifying that a particular gait pattern (or sequence of steps) sums up to a viable and stable pattern for walking.

As stated in Chapter 3, the nodes of ELMA are configured such that software can be downloaded to them over the fieldbus. Despite the versatility of this it still takes some time (a few seconds to several minutes depending on the complexity of the program) to modify and download new programs. Therefore, initial tests were performed with a centralised leg positioning scheme since only one node needed modifying. This was used to verify the static stability of the robot in various positions and to calculate the leg extrema positions.

Since each leg node only performs signal-conditioning functions, there is minimal computational load on them. Equally the central brain only has to traverse a position table so it has little more to do than output new leg positions; something the network handles transparently. If range information provided by the ultrasonic sensors described in Chapter 3 is fed into the controller then the computational load on the brain rises since it must decide upon appropriate behaviours, modify gaits to take avoiding action, and update the leg positions.

Resistance to perturbations is low since the leg nodes are incapable of operating without the brain. Failure of the brain node or network connections to it leads to complete incapacity of the hexapod.

4.2.2 Centralised Sequencing

This control architecture is a simple advance on the previous one. The individual state tables are moved out onto the legs, or *'offloaded'*, (WENN94). The brain node outputs a sequence count that is used by each node to index into its own local state table. As mentioned in Chapter 3, Section 3.6, each leg needs to add a phase shift to the index count in order for the possible gait sequences (ripple, pair or tripod) to operate correctly. If this were not done, all the legs would be in the same part of the sequence at the same time resulting in them adducting simultaneously and causing the robot to fall to the ground. The phase shift is generated by naming each of the leg nodes and then using this number to adjust the indexer.

Allowing the nodes to self identify using configuration information permits the program code on each of the nodes to remain identical thus allowing rapid code development. In addition, since the leg nodes could theoretically be swapped for one another, they can be said to be *interoperable*, as described in Section 2.4.6, at the most basic level. It is useful to note that the polarising of a node according to its physical position does not prevent the nodes basic functionality. In other words, the leg would operate in isolation without this information. Only when it is installed in a network does it become necessary to provide this data. The program function that performs translations based upon each legs location can be considered part of the node's *Performance Model.* The veracity of this statement is due to the fact that the Performance Model provides operational behaviour in the absence of specific information. In this case, without the polarising function a different program for each leg would be required with appropriate phasing predefined.

4.2.3 Distributed Positioning with Centralised Scheduling

The final non-reactive structure considered is created by modifying the previous algorithm so that it no longer outputs a discrete index. The output value is instead treated as a *heartbeat* signal. Individual leg nodes have their own state table, as before, along with a localised indexer into each table. Whilst this is only a simple modification to 4.2.2 it provides the basic functionality upon which the following reactive controller is based.

4.3 Reactive Gait Control

Reactive controllers use sensory stimulus and response reactions to produce leg motion and gait co-ordination, FERR95. As described by Arkin, ARKI98, these systems have the following properties:

- Low-level motor reflex responses form the basic behaviour by combining sensor and actuator units at a primitive level.
- Abstracted representational knowledge is not required to generate a response. This is advantageous in hazardous or unknown environments where

time consuming abstraction and sensory interpretation might delay robot-preserving tasks.

- Animal behavioural models are often used as the basis of these controllers. As mentioned previously robots provide observable platforms for testing theories on the behaviour of animals. Equally, animal behaviours provide good models for developing efficient control systems, BEER92, CRUS94 and WEID93.
- Reactive systems are inherently modular from a software design perspective. Since they are based upon simple behavioural blocks, software can be built up in a series of layers without discarding older blocks.

The last point is of particular use when designing software for use in a distributed control system since small functional blocks can be validated on the robot hardware without need for simulation. Moreover, functional units can be included that allow for the generation of sensory information given partial sensor loss or due to a lack of available sensors.

The reactive gait control used on ELMA is based upon state machines. Other forms of reactive control do exist that are based upon neuroscience and ethology backgrounds. One such example is Beer's *Periplaneta Computatrix,* which claims to implement most of the high and low-order motor and goal seeking behaviours of the common cockroach using an evolved neural network, BEER93[1]. It is not the intention of this work to investigate these other avenues since they invariably place heavy demands upon the processors and are consequently unsuitable for real time operation within an autonomous mobile robot that uses fieldbuses for control.

4.3.1 Subsumption Architecture

Rodney Brooks proposed a behavioural scheme called the subsumption architecture, BROO86. In this reactive scheme, behaviour is built by adding layers of control on top of existing simpler layers. This is different to the more traditional method of task decomposition where linear sequences of operations are performed on the sensed data finally generating motor drive signals. Whilst the robot is processing the data and arriving at the final drive signals it is effectively blind, KELL97. With a subsumption based architecture the lowest levels of control have the highest priority, see Figure 4-1.

[1] Although this example exhibits all of the gaits discussed by WILS66 it is interesting to note it is still not a complete simulation since the cockroach has an additional fleeing response that entails raising the hind pair of legs, resting its thorax on the ground and only moving forward under the control of the two forward pairs of legs, HILL67. This results in a faster cycle rate since only four legs need to cycle in sequence.

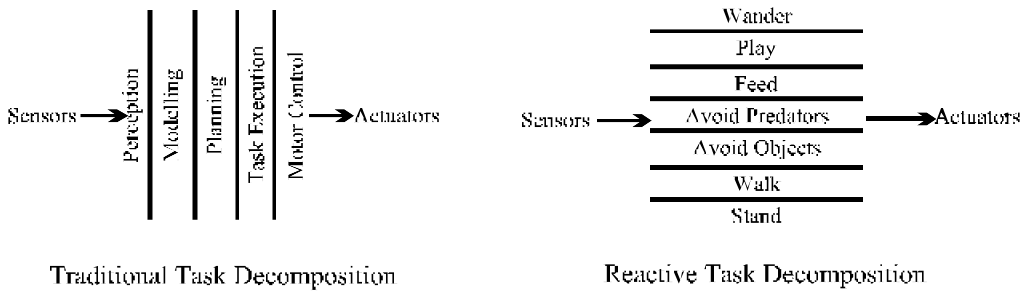

Figure 4-1 Task Decomposition

For a hexapod, the most basic level of functionality is to stand and this forms the bottom layer of the subsumption architecture. On top of this, the control structures necessary for walking are added. These modify the standing behaviour and cause the legs to cycle through a series of motions resulting in the robot walking. All the time the walking behaviour is operating ELMA still performs the necessary operations to prevent it falling over. As can be seen in Figure 4-1 additional layers can be added to enhance the overall behaviour. At the time of writing, the *Avoid Objects* layer has also been added. As has been stated ELMA uses a modified state machine called an augmented finite state machine to control walking.

4.3.2 Augmented Finite State Machine

The augmented finite state machine (AFSM) uses a regular finite state machine with additional registers, clocks and combinatorial circuits to yield a robust layered control system, BROO89. Additional AFSMs are added to form extra behavioural layers on the robot. In addition, the modular approach inherent in this reactive control system allows for neat encapsulation of the different parts of the control system at the leg node level. As the concept of a Local Object Model is extended, the functional model can be used to hold the AFSM for missing or incomplete parts of the robot structure. As was stated in Chapter 3 this permits the creation of software blocks that mimic the continuous motion of the servos although they actually operate in a closed loop fashion and do not provide positional feedback. Additional blocks can be added that simulate the value of a ground contact sensor. Whilst this is unused for walking it is useful for tests that require introducing lesions to the robot structure and then monitoring behaviour.

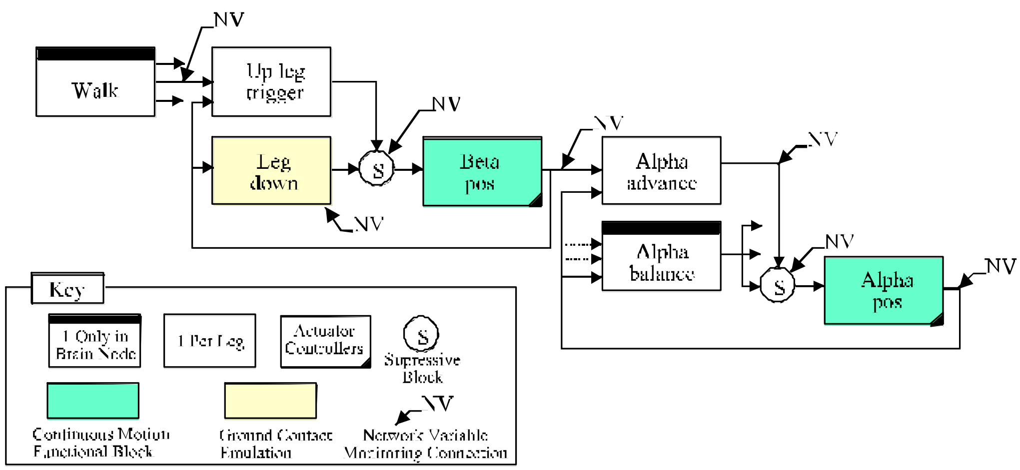

Figure 4-2 AFSM as Applied to ELMA Hexapod

Shown in Figure 4-2 is the AFSM model for one leg of the ELMA hexapod. The operation of this controller behaves exactly like the one in Genghis with certain additions to cater for the hardware specifics of ELMA and the *LonWorks*® network. It operates in the following manner:

1. The *alpha pos* unit drives the *alpha* angle servo controlling retraction and protraction. The *beta pos* unit drives the *beta* angle servo controlling abduction and adduction. At power up and without any other intervention, the servos are driven so that the robot is standing with its legs centralised. From the perspective of the leg node (and network) the servomechanisms operate in a closed loop fashion and so an additional routine takes the command input and drives the servo slowly to the final position providing

emulated continuous motion. Whilst this is occurring the current interpolated position is output as a network variable.

2. The leg down machine is driven by another simulation unit within the performance model that monitors a pseudo ground contact signal. At any time the leg is not on the ground it generates a signal to the *beta pos* unit to set the leg down. The performance model of the ground sensor uses *alpha pos* and *beta pos* data to determine whether the leg would in fact be on the ground at that part of the cycle. Furthermore, the ground contact signal is provided as a network input value so that it can be driven externally from the node to induce lesions in the controller.

3. An *alpha balance* machine is added to the brain node that monitors the *alpha pos* value of each of the legs. These values are summed and normalised such that central positions have zero effect, forward positioned legs have a positive influence and those positioned towards the rear have a negative influence. The *alpha balance* machine operates at 25Hz continually generating new values to adjust the position of each leg so that the robot remains in static equilibrium. The interconnection between the brain and six leg nodes requires the addition of twelve network variables each of which is potentially updated at the 25Hz rate.

4. Each leg has an *alpha advance* machine added to it. Whenever a leg is not in contact with the ground (as dictated by the current value of the *beta pos* machine) it drives the leg forward and suppresses the signal coming from the *alpha balance* machine. Since the *alpha balance* signal is suppressed the leg moves ballistically forward and at the same time the other five legs are commanded to move back slightly by the *alpha balance* machine to compensate for the new centre of gravity. This basic effect allows the robot to advance its position.

5. Next an *up leg trigger* is added to each leg. This machine monitors the *beta pos* of the leg and when it is down and a trigger signal is received it suppresses signals from the *leg down* machine and directs the leg to adduct.

6. The trigger source is provided by a single *walk* machine in the brain node that executes a set of predefined sequences amounting to the various hexapod gaits.

With these basic AFSMs in the leg and brain nodes, the robot is able to move forward. In the original control system of Genghis, additional state machines were added that provided directional control, pitch stabilisation and an infrared source following behaviour. In ELMA these additional behaviours were not added by using additional state machines. Instead directional code was added by using a pre-set gait for turning and then selecting it when the obstacle detecting ultrasonic system directed.

The system implemented on ELMA verifies the applicability of the augmented state machine subsumption architecture for hexapod control. In particular it has proven to be a suitable candidate for implementation on the Neuron processor based nodes. The interconnection between nodes is easily handled by network variables. The layered approach to the control system with network variables at each nodal point (i.e. each connection between state machines) allows for easy expansion and reconnection of functional blocks at a level external to the node. This is consistent with the desires stated in Chapter 2, Section 2.4.6 whereby control system design and expansion has evolved to a state of just interconnecting functional blocks at a system level. Moreover, since the source of the connection signals can be external or internal to the node, additional computational complexity, not necessarily located on the node, can be added at a later date.

The next section goes on to cover the final version of software that has been tested on ELMA. This consists of a learning algorithm that is based on a set of rules and goes on to generate a path for the end effectors (or feet).

4.4 Leg Path Generation by Reinforcement Learning

As stated by Arkin, ARKI98, reinforcement learning is widely used in robotic systems for adapting the behaviour of a robot. It is numeric, inductive and continuous. The basic premise is that by applying a reward immediately after the occurrence of a response increases the probability of it reoccurring. Conversely providing punishment after an incorrect response decreases its probability, THOR11. Sutton and Barto provide an excellent introduction to the subject in SUTT98.

To create the required reward and penalty a component is required that evaluates the resultant response based upon current state, target and input condition, this is known as a *critic*. Maes and Brooks (MAES90) have applied this form of learning to the subsumption architecture already on Genghis. The Genghis *critic* in this case received exteroceptive data from two ground contact sensors added to the robot’s belly and a trailing wheel attached to the back. In this case, the task was to learn to walk forward.

Rather than adding additional sensors the *critic* used on ELMA relies on a rule set that varies depending upon which phase of the gait the leg is in. As such, it implements a Q-learning function as described by Watkins and Dayan, WATK92. As can be seen in Figure 4-3 two *critic* functions are required depending on whether the leg is retracting or protracting. Each has its own rule set but both operate to find an optimal path between the end-points during each phase of a step.

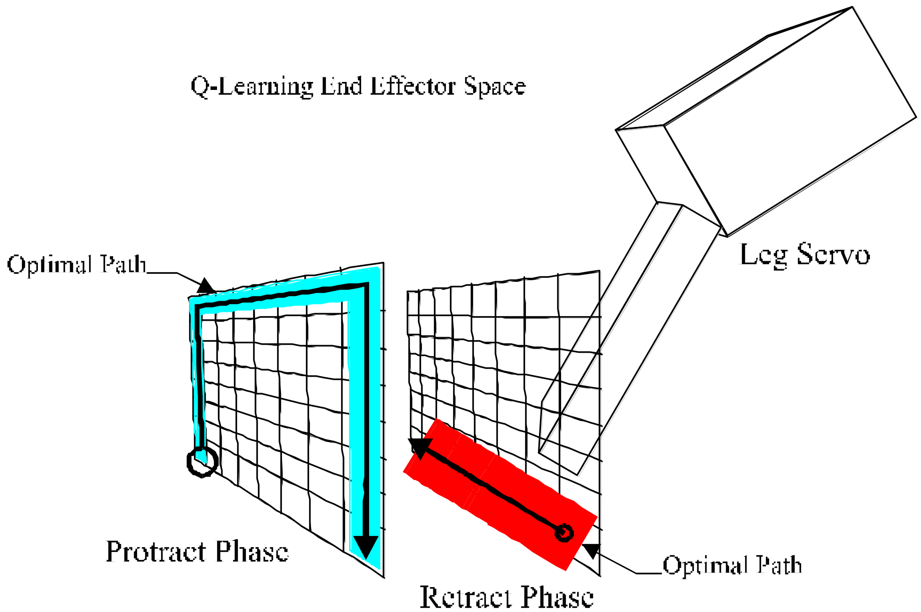

Figure 4-3 Reinforcement Function Spaces

Referring to Figure 4-3, the workspace envelope for the foot of a given leg is divided into a uniform grid of *tiles*, SUTT98. A uniform mapping was chosen since there was no specific need for irregularly sized *tiles* and it provided a direct mapping onto the servo control values. Starting at the lower left corner of the *tiling*, the target is set as the lower right corner for protraction. For retraction, the starting point and target are obviously reversed. For each *tile* within the workspace there are eight possible directions the leg can move. These correspond to the four orthogonal directions through each side of the *tile* and another set of orthogonal directions at 45° to the first set through each of the corners. The reinforcement rules are as follows:

<u>Retraction</u>:

- Moving rearwards (with respect to the body) is rewarded.

- Moving down is rewarded, and since the workspace is bounded, moves off the bottom of the grid are neither rewarded nor penalised.
- A move in any other direction is penalised.

<u>Protraction</u>:

- If the foot is in the left most column only vertical moves upwards are rewarded. Moves to the right (front of the robot) are penalised by a small amount. Any other moves are penalised by a large value.
- If the foot is at the right most column vertical moves down are rewarded. Further moves to the right are neither penalised nor rewarded. All other moves are penalised.
- If the foot is in any other vertical column, moves to the right and up are rewarded and all other moves are penalised.

Each leg implements its own set of workspaces. The drive signal forcing the leg to advance a step derives from a CPG in the brain node. At each step a new direction is chosen for the foot based on a weighted roulette wheel approach where previously chosen successful choices are given a higher probability of reoccurring, MITC94. After the move is executed the result is evaluated and a reward or penalty is applied to the chosen direction in the previous cell. This process continues until all the legs have arrived at their respective destinations for this part of the gait. The two end effector spaces are then swapped and the process repeated.

The reward and penalty values for both functions were identical and were chosen by evaluating the performance of the algorithm over a number of runs. It was found that too high a value could result in the leg *sticking* in a corner or at an edge point. Conversely, too low a value would cause the leg to take a long period to settle. Consequently, over a number of trials, a reward value set at 0.12 of the normalised probability range and a penalty value of 0.08 was found to offer a suitably short convergence time and good resolution. To analyse the performance of the algorithm and to determine the time required to settle the following approach was used:

- Each *tile* within the *tiling* is assigned a visit count starting at zero.
- When the *tile* corresponding to the current foot position is entered, the visit count for that square is incremented.
- The learning sequence is executed a number of times and at discrete points within the cycle (after 1 step, 5 steps, 10 steps etc) the visit counts for the entire *tiling* are recorded.

The visit count matrices were plotted as a three dimensional surface with the base of the surface corresponding to the grid of *tiles* (i.e. the grids seen in Figure 4-3) and the height corresponding to the visit counts at that moment within the learning sequence.

Shown in Figure 4-4 and Figure 4-5 are the surface plots for a sequence of 1000 steps. This corresponds to approximately 25 minutes of walking time. After one step, the irregular surface indicates that the *tiles* are visited on a random basis due to the equal probability of moving in any direction from a given *tile*. After some five steps, the learning algorithm has started to effect the end effector position favouring moves along the ground for retraction and high ‘horseshoe shaped’ moves for protraction. After 10 steps, the final form of the learnt end effector trajectory is forming and after 50 steps, there is little wandering outside the path prescribed by the rules. It is useful to note that the learning algorithm favours a diagonal path from the lower left corner to the upper right corner when protracting. This is due to the rule set rewarding equally right and upwards moves whilst operating in the central columns and so initially biasing the trajectory.

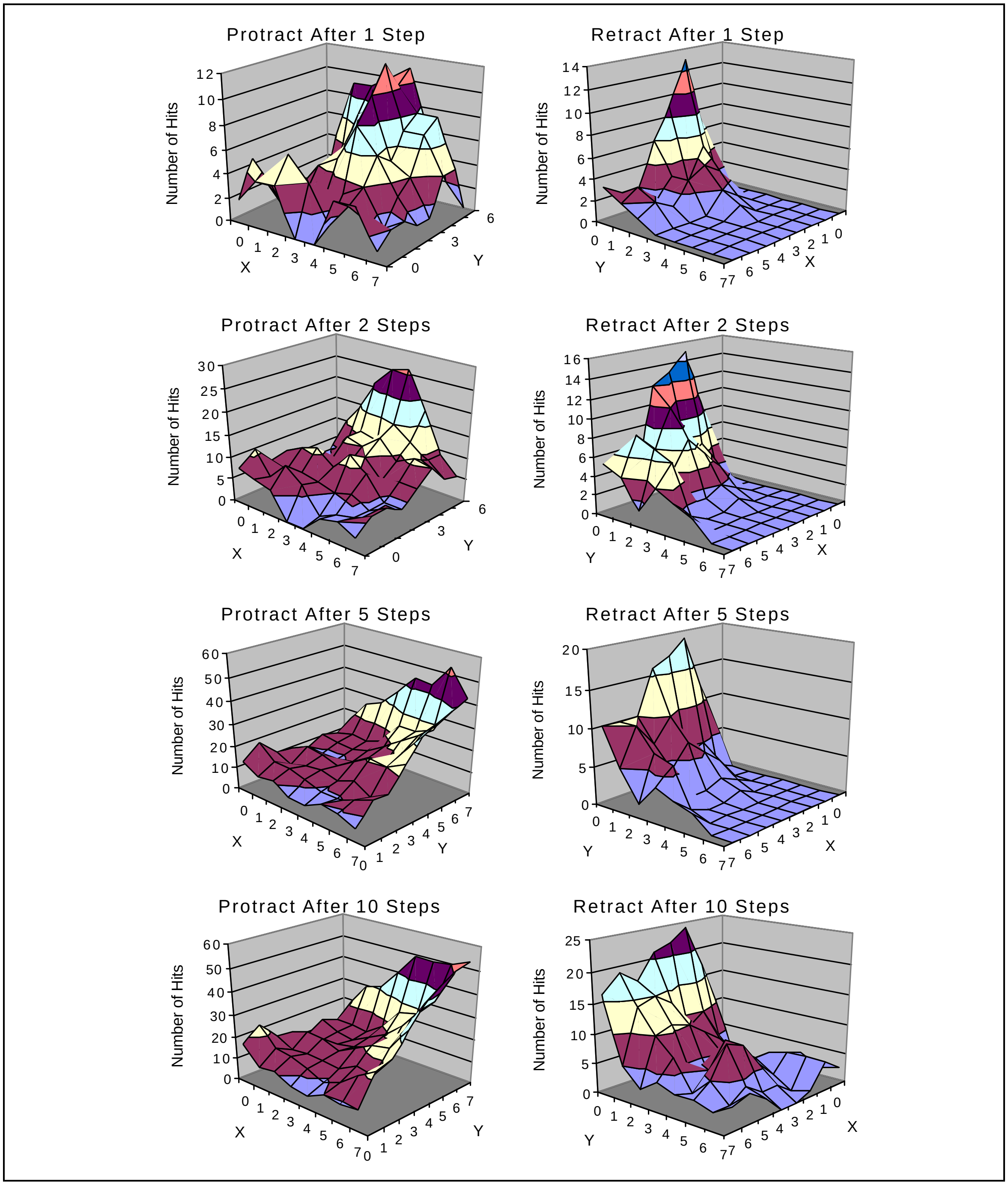

Figure 4-4 Traversal Count for Steps 1-10

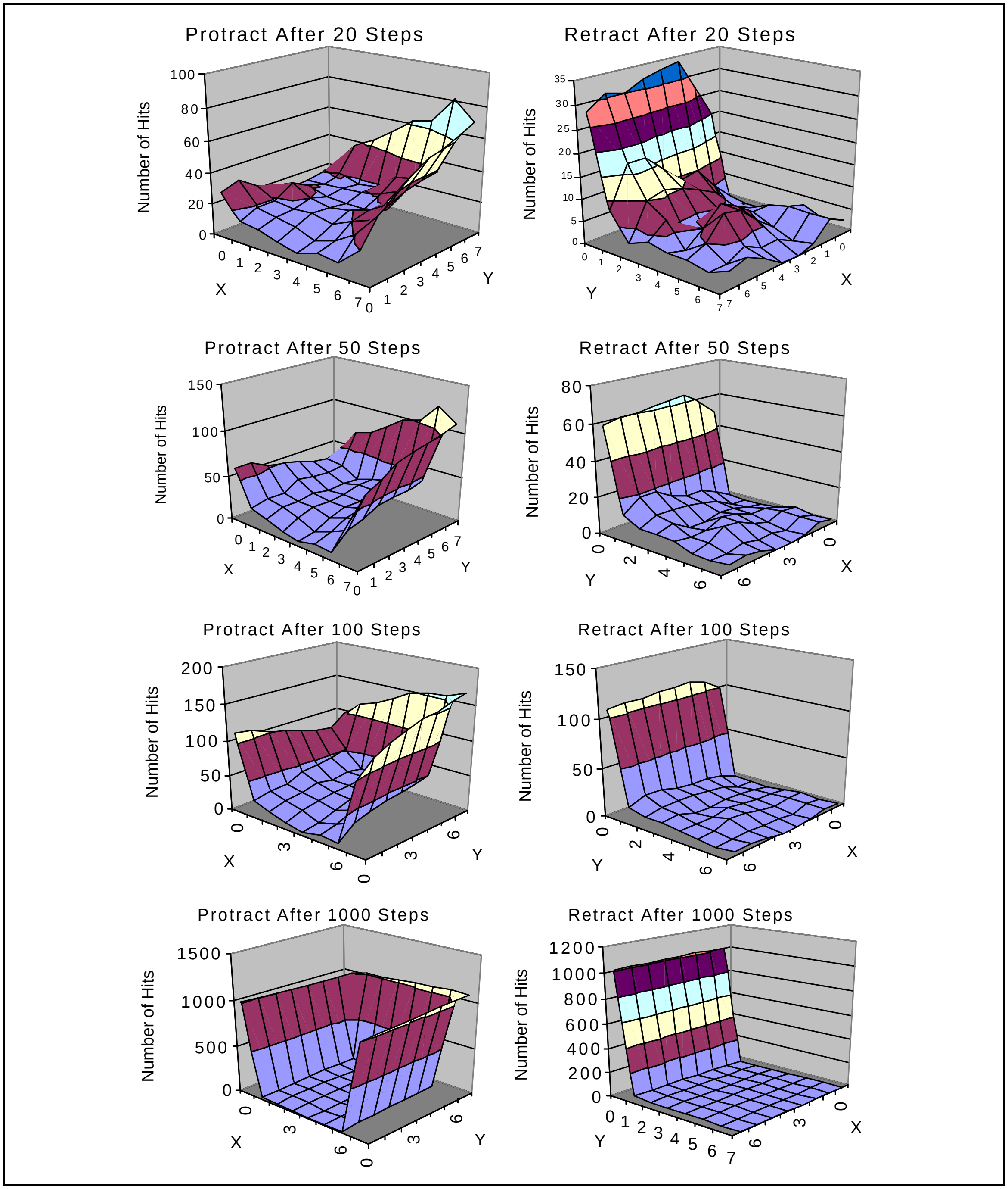

Figure 4-5 Traversal Count for Steps 20-1000

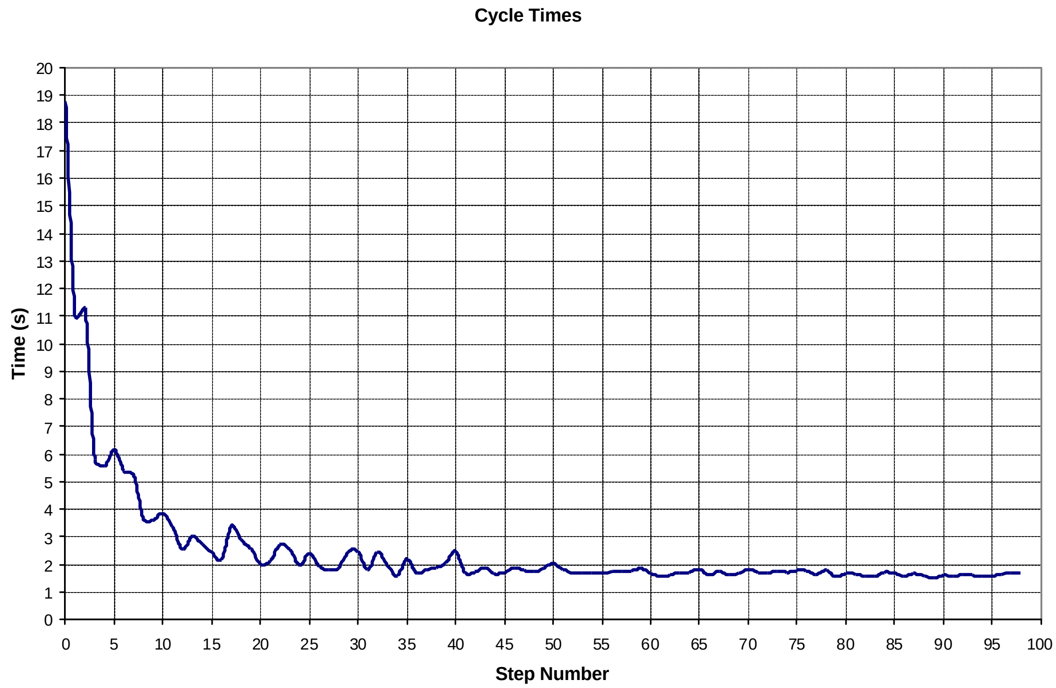

Figure 4-6 Average Leg Cycle Times Over Five Test Runs

To determine the speed with which the algorithm converges, the time taken to perform each complete step (as defined previously) was recorded. This was then plotted against the step number as shown in Figure 4-6. The optimal time for the chosen paths is 1.55ms; this is defined by the step interval pattern generator in the brain node which operates at 18Hz. By analysing the graph, it can be shown that the cycle time standard deviation is 0.96 after four steps and it has reduced to 0.097 after 50 steps. Qualitative analysis of the hexapod when operating under this software architecture shows that it has settled down to a stable pattern of co-ordinated leg movement after approximately 10-15 minutes of operation. Whilst this time corresponds to over 400 steps this lengthy period was due to dynamic instabilities in the robot when one or more legs took longer to settle consequently increasing the load on the remaining supporting legs. The increased loading caused vibration that was, initially and incorrectly, attributed to algorithm instability.

For real-time evaluation of the properties of the learning algorithm a number of network variables were made available that provided access to the learning parameters. A *tiling* of 8x8 with eight possible directions per *tile* leads to 512 discrete values. However, the *tiles* at the edge of the grid do not store parameters that would take the leg out of bounds. Equally, the corners omit those values representing moves off the adjacent edges. The total minus the edge and corner adjustments still yields some 420 interconnection probabilities for just one phase. The two phases, retract and protract, also have the *visit count* value associated with each *tile* so the total amount of data is 968 discrete values. This amount of information cannot readily be presented to the network in its native form (the Neuron processor has a limit of 63 network variables). Instead an indexing method was used whereby values for specific *tiles* could be requested and then read from one network variable. Furthermore, whilst the robot was learning the most recently updated value was also provided from the same variable.

4.5 Comparison with Existing Controllers

At the end of Chapter 3 experimental results were shown for the bandwidth usage of the robot whilst under the control of a simple state machine controller. The bandwidth usage for the single step gait is shown graphically in Figure 3-15 whilst that for the tripod gait is shown in Figure 3-16. Using this data as a performance baseline, the controllers shown in Section 4.3.1 and 4.3.2 are compared. When considering the controllers used in other published hexapod material, such as BROO86, FERR95, MAES90 and WEID94, it is apparent that there is very little quantitative data available on control system performance.

This is primarily due to the differing backgrounds of the research teams (i.e. computer science, psychology, biology or ethology) and the different tasks for which the robots were built. Despite the lack of a comprehensive set of performance figures, it is possible to ascertain some common factors either by examination of the published material or by extrapolation from video material[2].

Considering the figures 3-15 and 3-16 it can be seen that the average packet rate for the single gait is 42 packets per second whilst that for the tripod gait is some 80 packets per second. These numbers translate to bandwidth usage's of 7.4% & 15.1% respectively. Whilst under the control of the simple state machine both figures show characteristic increases in bandwidth usage around the time that new servo positions are sent to the leg controllers. The relative performance of the two controllers explored in Sections 4.3 and 4.4 can be tested against this data.

4.5.1 Performance using AFSM

Controllers based upon an augmented finite state machine have been used on a number of different hexapods, FERR95. The one used with the distributed control system on ELMA was written especially to test the capabilities of the *LonWorks*® network. Equivalent data to that obtained from the centralised controller in Chapter 3 is shown in Figure 4-7. It can be seen that the bandwidth usage during the retraction and protraction phases of each step averages 3.1%, approximately half that of the centralised controller. Furthermore, the worst case

[2] For instance, there is a published minimum leg cycle time for MIT's Genghis robot and this can be verified by comparing it with a video sequence that can be found on the World Wide Web.

usage only rises to 6.3% during the abduction and adduction phases when high velocity moves (implying many more network updates) occur.

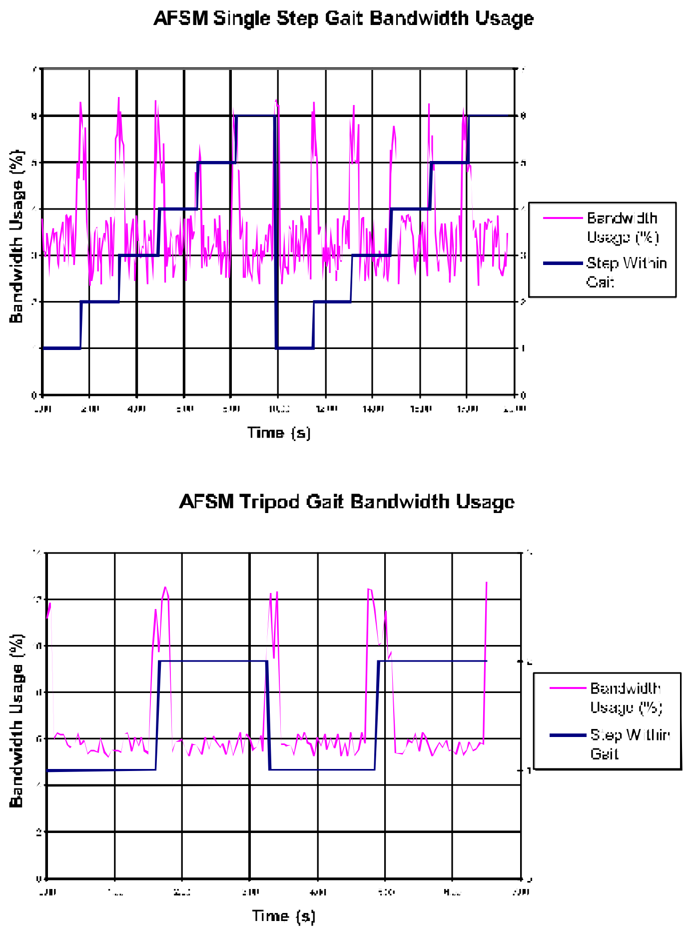

Figure 4-7 AFSM Bandwidth Usage

Brooks states, in BROO89, that the AFSM controller used within Genghis has both the single step gait and tripod gait programmed into its sequencer. Brooks goes on to state that both gaits complete a full cycle every 2.4 seconds. In the case of the tripod gait simultaneous *lift triggers* are sent to triples of legs every 1.2 seconds whilst for the single step gait a *leg trigger* is sent to a different leg every 0.4 seconds.

Considering the step time shown in Figure 4-7 it can be seen that the single step gait used on ELMA does not compare too favourably with the Genghis one since it has a cycle time of 9.8 seconds. This is primarily due to the slow response time of the sequencer, which also has to perform the inherently slow task of interpreting sensor data from the ultrasonic eyes. However, the 2.9-second cycle time for the tripod gait compares much more favourably with that of Genghis.

4.5.2 Performance using Leg Path Generation

Shown in Figure 4-8 are the bandwidth usage figures from ELMA when under the control of the learning algorithm. It can be seen that the usage is significantly higher than that for the reactive (AFSM) controller. This logically follows from the need to propagate many more network variable values between the legs and to the centralised balance controller.

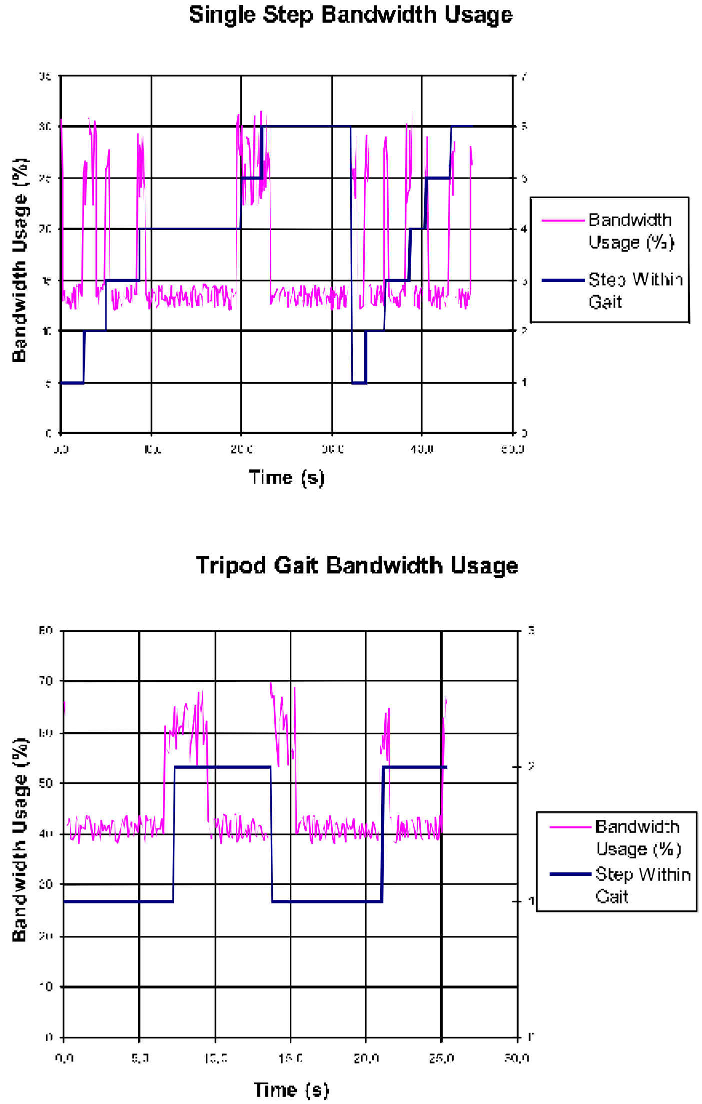

Figure 4-8 Reinforcement Learning Bandwidth Usage

Comparison of this type of learning controller with other published material is difficult due to the differing control strategies. For instance, a Genetic Algorithm based approach (as seen in the Stiquito robot of Parker et al, PARK97) will typically yield data that is only suitable for comparison with other GA based solutions.

However, Kirchner has implemented a comparable controller on the Sir Arthur hexapod, KIRC97. Here a Q-Learning function was used to learn complex behaviours at both the leg trajectory level (as in ELMA), the leg sequencing level (or gait generation) and finally at the goal achieving level. The results from the leg trajectory level of Kirchner's implementation show comparable results to that obtained from ELMA and shown in Figure 4-6. Shown in Figure 4-9 are the cycle times against step number within the learning cycle. It can be seen to have a similar characteristic curve to that of Figure 4-6. The settling time for the ELMA is marginally better than that for Sir Arthur, taking only 50 steps compared to 80 steps for the latter.

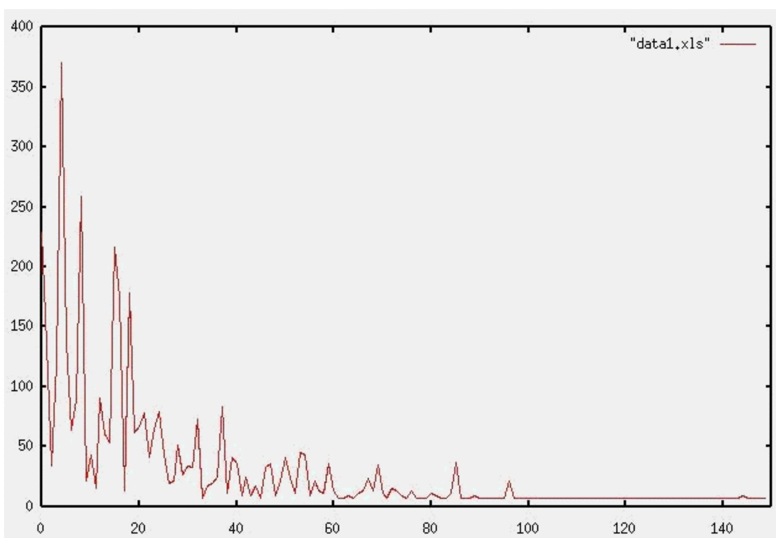

Figure 4-9 Q-Learning for the Sir Arthur Hexapod (reproduced from KIRC97)

The ELMA robot discussed in the section is one example of a distributed control system with many identical and complex nodes. The whole concept of such networks is however much broader and many application areas exist. In the next Section another class of such network is considered whereby many distributed controls are used within the built environment. Like the ELMA robot, these nodes need managing and visualising and at this point it is also convenient to explore the Local Object Modelling concept used to aid this task.

4.6 Local Object Models

The problem with gathering internal state data from a node using the method employed on ELMA is that a general purpose monitoring tool, such as Echelon's *LonMaker*™ (ECHE98b), or IEC's ICELAN-G™ (IEC98) have predefined methods for displaying network data that are not easily adapted to such application specific demands. The solution to this problem adopted by the OLE for Process Control organisation (OPC98) is to provide device specific controls. These allow a suitable network monitoring tool to activate device specific objects (or 'Plug-Ins' in the Echelon LNS system) that are written by the supplier of the device. These objects are instantiated at program run-time and display the device's data in a suitable format. However, the availability of both this device specific software and network management tools to support it is problematic and, in the authors opinion, will remain so for the foreseeable future.

In Chapter 2, Section 2.5, the Local Object Model (LOM) was introduced. The LOM is a distributed communication structure that contains several classes of model that can be implemented across nodes in a network, FOST98c. By designing nodes to handle these models, this allows a user to access a variety of network services at a local level. These services range from simple interaction with the control devices at the user's location to sophisticated network management and access methods. This Section goes on to explore the specifics of the LOM and it's various sub-classes of models. In particular the Localisation Modal class is shown as a useful extension to any network device that allows easy visualisation of its physical parameters. By implementing a subset or all of the LOM models, modular components can be created that allow representation and interaction with the large amount of information in a distributed control system. This has been demonstrated, in the first case, using the ELMA robot. The LOM technique was not initially applied to the hexapod having instead been developed to support various networked facilities in the built environment. Therefore, it is constructive to consider its background before examining the extensions to the model that aid visualisation of systems such as ELMA.

4.6.1 Background

From very meagre beginnings, the use of electronic systems within the built environment has reached the point where market acceptance is widespread and they are becoming all pervasive (FERR95). Today's modern home would not be complete without its ensemble of computers, VCRs, televisions, telephones, heating, security, and light controllers. Whilst, at the time of writing, these devices are not normally connected, the time when they will migrate to one common household network of devices is fast approaching, GROV95. Energy suppliers are also investigating '*smart homes*' that will use fieldbus networks to connect assorted devices providing monitoring and demand side metering facilities. Indeed domestic appliances are already available that connect to energy producers and carry out a negotiation for the most economical electricity, INTE99. Energy suppliers see the potential for vast savings from being able to accurately forecast loads, KIER97.

Networked control systems within a building are a solution to the problem of interconnecting all the disparate devices and systems thus aiding installers and suppliers. However, for the end user these same networks can offer considerable advantages. The cost of running individual homes is reduced, energy efficiency is increased and a whole range of new possibilities is created for the control of items around the environment. In particular, this technology has been identified as particularly suitable as a base for developing technologies to support independent living for elderly and disabled users, COOP96. The Department of Cybernetics has been involved in distributed systems research for a number of years and one of the early projects led to the precursor to the Local Object Modelling technique.

4.6.2 HS-ADEPT

Supported by the European Commission Telematics Initiative for Disabled and Elderly people (TIDE), the *Home Systems – Access of Disabled and Elderly People to this Technology* (HS-ADEPT) was aimed at developing networked technologies to aid disabled and elderly people. There are a number of existing *smart home* systems already in existence, such as the *LonWorks*® based TIPI (LAPO95), but they are not directly aimed at TIDE's target users.

At a systems level a distributed control system installed within a building operates on the environment that contains it. The HS-ADEPT project was aimed at giving its users the ability to interact with networked control devices in their locale. From both the control network and system installers point of view there is normally minimal or no need for localisation of any device. However, to enable localised control for a user situated within the plant information about local objects is required. To support this a concept called the Local Device Model (LDM) was developed as described in FERR95. The LDM holds local information about the control facilities available from a limited number of nodes in the immediate local area. For the HS-ADEPT project, the LDM was implemented as a specialised node within the Home Systems network known as the User Interface Feature Controller (UIFC).

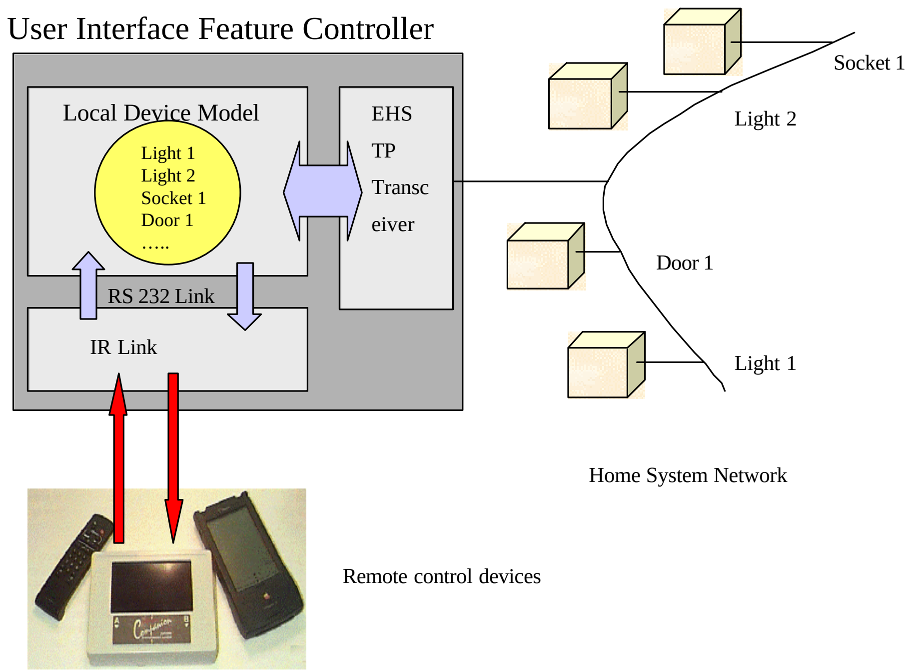

Figure 4-10 HS-ADEPT UIFC Block Diagram

As shown in Figure 4-10, the UIFC exists as a custom node within a network of Home System based nodes. The devices within the Home System network are typically connected to simple devices such as light switches or door openers. The UIFC is *taught* (by using a special configuration mode) which devices are in its local area. Because of the strong typing and simple objects that exist within the Home System specification (EHSA92) this gives the UIFC sufficient information to interact with its learnt devices.

When operating normally a user is given a remote control device that communicates with the UIFC over an infrared link. This remote control handset allows a particular local device to be selected after which a list of actions pertinent to that device is displayed. The user can then select the appropriate action and observe the result. Additionally messages can be sent down to the handset giving the user useful feedback. The issue of designing man-machine interfaces is a large area, PREE90. The complexity increases when special considerations such as disabled or elderly users need to be taken into account as discussed by PETR98.

A variety of different remote control devices were tested with the UIFC ranging from simple single push button handsets on to sophisticated Apple Newton PDAs. As a related undergraduate project Hammond, HAMM96, designed an augmented reality system overlaying information from the UIFC onto the users view. A user could select highlighted objects and control them via the UIFC. This system forms a very basic example of the work developed by the author in Chapters 5 and 6. The information presented to the user was restricted to overlaying *hot spots* on controllable features. There was no synchronisation between real world devices and their related virtual features within the view.

The HS-ADEPT UIFC functions on a basic level gathering and presenting information about local nodes. Unfortunately, this user interface requires the addition of an extra node to the environment. Moreover, the LDM it contains is restricted to relatively simple devices. Whilst the LDM concept is valid, it suffers from being somewhat inflexible and unsuitable for further expansion.

Within HS-ADEPT the nodes local to the UIFC do not contain modelling information themselves. Instead, the UIFC has to be taught about its local devices and generate the control models for itself; this restricts the flexibility of this control paradigm. The basic LDM concept has been greatly expanded in a new project; covered in the next section.

4.6.3 ARIADNE

Again funded by an European Commission TIDE grant, '*Access, Information and Navigation Support in the Labyrinth of Large Buildings*' (project no. DE3201) seeks to support disabled and elderly users gain access to large buildings. Named after the Greek Princess, Ariadne, who helped Theseus navigate the maze of the Minotaur, ARIADNE supports a range of navigation and information services for its users.

The technical aspects of the project can be seen in FOST98d. However, in summary, ARIADNE is based around installing a network of connected nodes around a building. This may mean retrofitting to an existing building system of lights, HVAC and security or installing from fresh. Nodes within the environment are connected together both physically and logically. Agents sent around the network are able to perform a variety of searching and navigation functions such that users can obtain information about their location, other object's locations, navigational information and perform control actions on connected devices.

Unlike the HS-ADEPT UIFC design, local device models are now stored on the devices and so the enhanced LDM used by ARIADNE has been renamed 'Local Object Model' to distinguish it. The LOM has been broken down into a number of different types that can be implemented across various devices, see Figure 4-11, and these are detailed below. It should be noted that not all model types would be implemented on a specific node. In particular the full range has only be implemented on the *Access Node* designed by the author and used within the Ariadne system.

Shown in Figure 4-11 is a conceptual diagram of the LOM. The following sections go on to explore the various classes within the LOM, and how they can be used to represent and interact with a fieldbus.

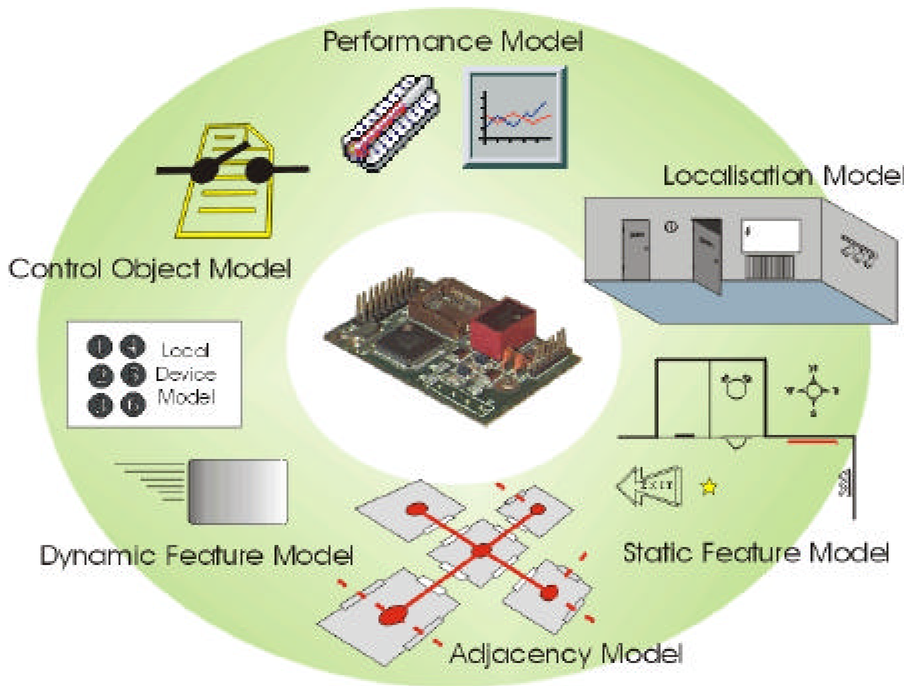

Figure 4-11 Local Object Model (LOM) example

4.6.4 Performance Model

In the first part of this Chapter a method of encapsulating functions to simulate incomplete data or process states called the Performance Model was introduced. The idea of using a Performance Model can be readily extended to cater for the various dynamic properties of a system. For example, one aspect of such a model might be a Kalman Filter Algorithm that generates process state estimates for the node in the advent of partial nodal or network failure; Grewal and Andrews provide a good explanation of Kalman Filtering in GREW97. Durrant-Whyte et al. have developed a Distributed Kalman Filter that is particularly suitable for networked implementation (DURR90), although it places a high load on the network bandwidth due to the amount of state information that needs transmitting. Glover et al. (GLOV98) have developed Sensor Validation techniques on Neuron Processor based nodes where the validation and calibration of sensors and actuators are external to the actual process loop and could be conveniently encapsulated within the Performance Model. It is not within the scope of this work to examine all of the possible software structures that can be placed within the performance model.

4.6.5 Control Object Model

The Control Object Model is the set of functional blocks presented at the application layer to the system installer. In the HS-ADEPT project these control models are constructed within the UIFC based on variable type information[3] obtained from the physically adjacent devices in the network. Consequently, they have only a logical association with the devices to which they refer.

For the ARIADNE project based on the *LonWorks*® system, these models are encapsulated within the devices themselves. The *LonMark*™ Organisation defines an assortment of control object types that allow for interoperability between devices and user interfaces. Shown in Figure 4-12 is a typical control object model for a node. The objects within the model can be accessed via the Node Object. Depending on the node, various sub-objects are then contained within the Control Model. Configuration of the objects within the model may require many items of data. Normally these are set by configuration network variables. However, the node limit of 63 network variables may pose a problem if a lot of parameters are required. Consequently, a file transfer mechanism exists that allows the mapping of node memory to a virtual *disk space* that can then

[3] A variable within a computer program has a *type* which describes how much storage it takes and how the storage is delimited. For instance, the 'C' line ***int I;*** defines a variable ***I*** of *type* ***integer***. Amalgamations of existing types form structures and their *type* is their name. The interested reader is directed to any programming reference manual for a fuller explanation.

contain the configuration data. It also may be the case that communication between nodes requires block data transfers and so the file transfer mechanism is also used in this case, PHIL97. This is done in small packet slices of 32 bytes so that the network, already optimised for small control packets, is not swamped by large messages.

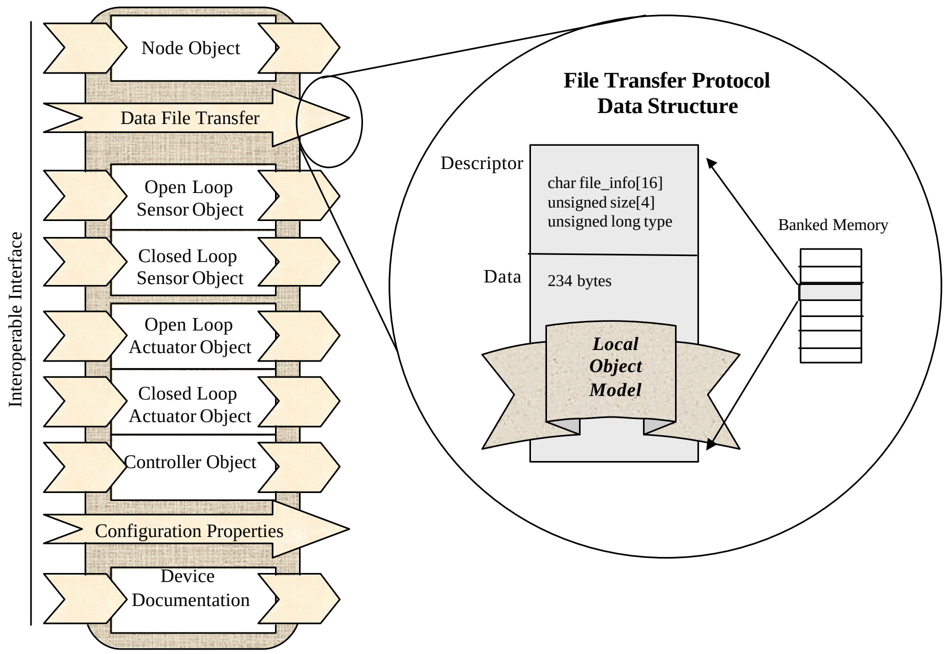

Figure 4-12 Control Object Model using *LonMark™* Interoperable Architecture

The later work contained in this thesis relies on it being possible to transfer and store relatively large, for a node, amounts of data (~20Kb), to support graphical and behavioural items. The file transfer mechanism is ideally suited for this application since it is already heavily specified by the *LonMark™* organisation and should be supported in some form on most commercially available nodes, PHIL97, LONM97.

4.6.6 Localisation Model

The Localisation Model represents the physical environment around the node. For nodes in the built environment this would typically be some part of the building related to the node, for other nodes it may be a feature near the device defining its position in plant relative terms. The localisation models can exist in many forms depending upon the target audience for the data.

For example, the TMR MOBINET programme is a collaborative effort researching methods of supporting mobile robots within hospitals and other health care environments. Since the robots envisaged by this project operate predominantly within a 2-dimensional workspace, height information is seldom needed. Consequently, floor plans or other *flat* maps may be sufficient to help guide the robot, OHAR98. It is useful to note that the localisation model provided to a robot need not be a map suitable for human interpretation. Instead, abstract data such as magnetic anomalies or light-meter readings can be used to distinguish orientation and positional information for the robot.

For a human user of the ARIADNE system the localisation model takes the form of a speech message describing the location. The *Access* nodes designed by the author for ARIADNE respond to appropriate user requests by playing the message so that they may orientate themselves to the environment.

Finally, the model may take the form a fully features 3-dimensional model of the surroundings, or part of the plant, around the node. When a user is presented with data in this form a variety of sophisticated interaction options become available. These are covered in Chapters 5 and 6.

4.6.7 Static Feature Model

The Static Feature Model contains features that are located close to the device, but are not electrically connected to it. Whilst they cannot be monitored and controlled, by being including in the description they can be located through the network. An example might be a telephone within a room that is near a room light controller. Each static feature is given a globally unique identifier within the environment.

Whilst the static feature is not directly controlled or monitored by the node it can still exist within the node's LOM. This allows for completeness in the representational aspects of the system and allows interconnection between the various classes of object. For instance, the door adjacent to a node can have a static feature associated with it and an entry within the node's localisation model. This allows for complete description of the environment in a compact and regular manner.

4.6.8 Dynamic Feature Model

Nodes within a network move very infrequently. However, it may be that objects that move around the environment need monitoring and so this model allows references to these to be stored. This allows the location and tracking of moveable resources such as people, equipment and robots.

For the ARIADNE system, users and equipment are provided with contactless ID tags that are monitored by microwave reader units placed around the environment. Originally designed for motorway tolling applications, the reader units are connected to the network by custom nodes designed by the author giving them a network presence. Subsequently, each is associated with a primary *Access* node in the environment that stores a list of the current tags within its domain, FOST98a.

Objects within a node's Dynamic Feature Model can be located via the network using searching agents, FOST97a. The agents used by ARIADNE present a useful method for accessing and obtaining information from a control network. The whole area of intelligent agents within computer networks is a rapidly growing research area as described by Wooldridge in WOOL94. Unfortunately, their implementation within a control network is limited by the inability to pass code between nodes (which languages such as JAVA permit). However, even limited data storing agents offer a novel alternative for *'data harvesting'* on a fieldbus system provided suitable search functions are implemented on each node.

4.6.9 Adjacency Model

The Adjacency Model contains a list of nodes that are adjacent to the host in the environment. It stores a record of physical connectivity between devices and includes a measure of the difficulty in translocating between the host node and the target.

Primarily used within the ARIADNE project, the Adjacency Model supplies the connectivity information required to allow searching agents to ascertain optimal paths between nodes. Shown in Figure 4-13 is the Adjacency Model for the Distributed Systems Research Group laboratory. An *Access* node in the centre of the lab contains five Adjacency models pointing to connecting nodes in the surrounding area. This allows a user to request navigational information and then be given a suggested direction leading to their eventual goal.

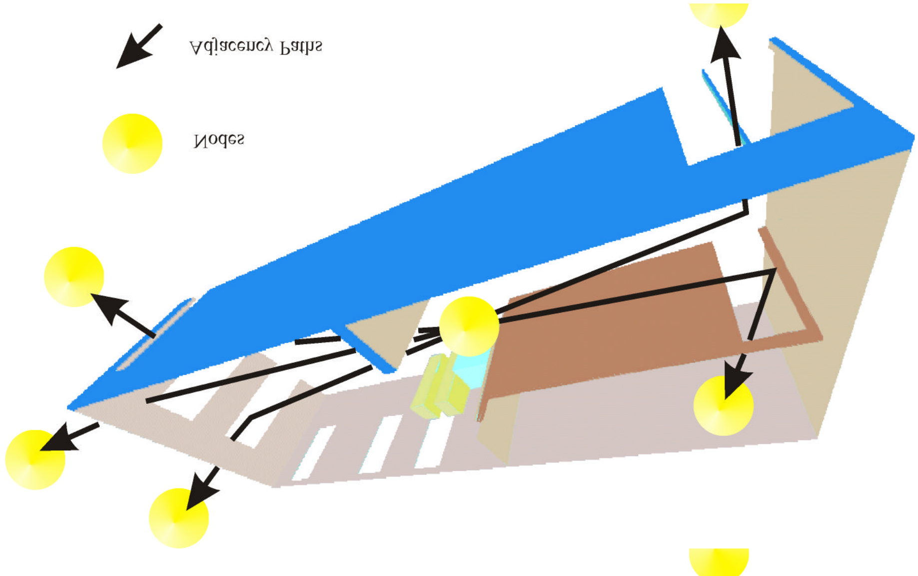

Figure 4-13 Adjacency Model for a Laboratory

4.7 Local Modeling Techniques on ELMA

Having covered the various features of the full Local Object Model this section goes on to describe how it can be used to aid the programming and visualisation of the ELMA robot.

From Sections 4.3 and 4.4, it can be seen that even the simple distributed control system used in ELMA can generate a large amount of information. For example, the subsumption architecture implementation generates over 80 pieces of data at 20Hz that are directly related to walking (servo positions, state machine values and so on), without even considering the network related parameters. It can be appreciated that a large plant will potentially have much more data.

4.7.1 ELMA Performance Models

Whilst designing the software for ELMA it has been stated that incomplete sensor information was emulated using functional blocks. It was also stated that these blocks could be said to form part of the node's performance model. This idea can be extended by adding additional performance models to completely emulate the state of the device. If this is done then real-time data can be fed into the performance model from the network at the same time as a simulation unit calculates new output values for the leg node. The question then arises, why is this useful?

It follows that devices written to be interoperable in this fashion permit their inputs, outputs and internal connections between functional blocks to be interconnected with external devices. Consequently, operation in the case of partial failure of a block (perhaps due to a sensor failing) can be emulated on a different processor. Moreover, complete emulation of the failed device can be performed by an additional replacement node. Obviously for this functional level redundancy to occur the program code within the emulation blocks must operate at the same level (i.e. processor instruction code level) as the code on the node. Considering Figure 4-14, the node contains the same program code as before which implements the subsumption architecture. Above this are a number of functional blocks that are implemented on the node. They operate in conjunction with the node level code to implement missing or incomplete functionality in the robot. Equally, they can be replaced by functional blocks lying outside the node itself. In themselves, these blocks form part of the node's performance model. However, to complete it an additional block is added that models the behaviour of the leg node using the remaining information from the node and data from the middle level blocks.

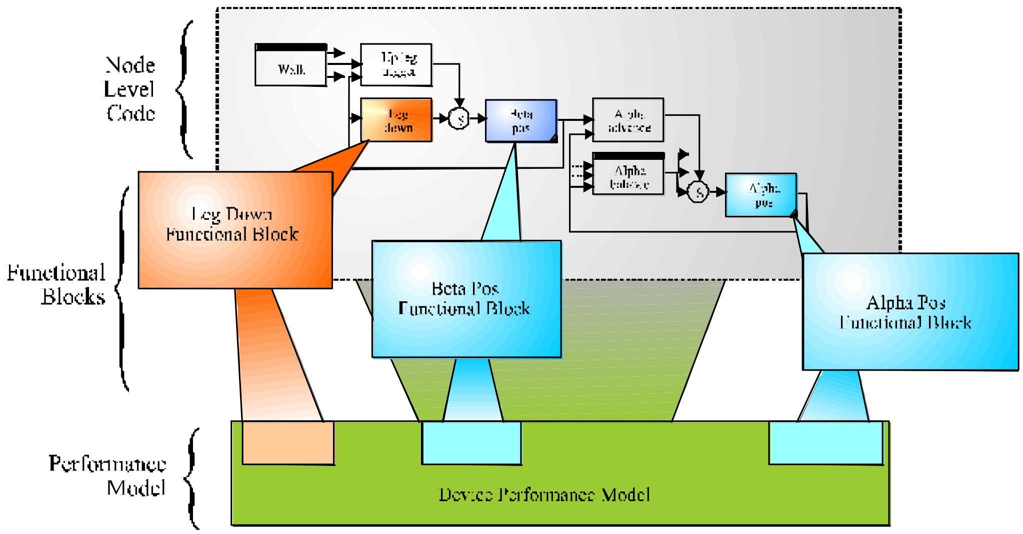

Figure 4-14 Leg Performance Model Decomposition

The complete device performance model on its own is an abstract quantity that can only usefully be duplicated on an equivalent Neuron processor based node. However, there is no requirement for the highest level model to be written in code compatible with the Neuron processor. Instead, the performance model can contain program code written for a completely different target processor. This may be some PC based program module or even a JAVA applet. Execution of the performance model on another host allows for software emulation of the device. If the host is then connected to the fieldbus via a suitable interface card then it can duplicate the actions of the node in real-time providing monitoring and control options. Obviously there are considerable constraints associated with storing program code designed for other hosts on a node due to its limited memory size. However, this can be overcome by storing a *virtual* reference to the performance model on the node and locating the actual model on the network server. This concept of abstracting performance models and *virtual* referencing is fully explored in the Damocles software covered in Chapter 6.

4.7.2 ELMA Localisation Model

The other LOM class implemented on ELMA is the Localisation Model. From Section 4.6.6 this model deals with storing a description of the physical area around the node. For the hexapod only two types of model are required. The head node deals with storing a description of the head and body of the robot. The leg nodes store a representation of the leg structure. For this application, the models were stored in a 3D representational format created using 3D Studio Max from Autodesk, AUTO96. The exact nature of this modelling format is covered in more detail in Chapter 6. For reference, the two types of model are shown in Figure 4-15, with the ELMA chassis on the left, consisting of head, body and eight supporting brackets for the legs (only four visible) and the leg node itself on the right.

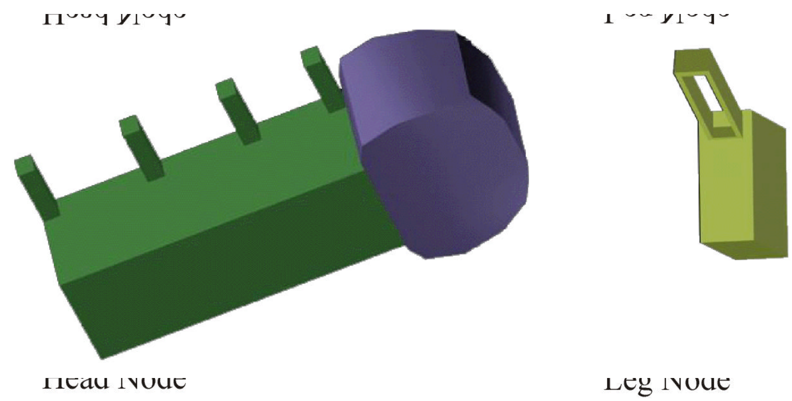

Figure 4-15 ELMA Localisation Models

A suitably written visualisation program can access the localisation model stored on each of the nodes using the file transfer mechanism previously described. Having obtained the model data the displayed images can be manipulated to display concurrent data from the robot as described in the next section.

4.7.3 Monitoring ELMA

During the course of this research, various programs were used to monitor and control the hexapod. The limitations posed by some of the earlier programs led to the generation of more sophisticated applications to facilitate easy operation. The desire to provide intuitive user interfaces to complex distributed systems is a natural one and many researchers are currently investigating this area. For example, there has been considerable research in recent times on advanced control room technology for power supply companies. Nakatani et al. (NAKA97) have developed a sophisticated tool that aids power-network operations managers to diagnose and interact with national electricity distribution grids. These interfaces show increased complexity and yet demand rapid responses from the user and so they must be made easier to use, LEAT95.

The basic network monitoring tools provided by *LonWorks*®, and most other fieldbus systems, are limited to pure textual representations of network variable values using a *browser*. These values are updated periodically and can be modified resulting in an update to the node's value. The numbers displayed are interpreted by the *browser* and scaled to the appropriate value using the typing information implicit in the network variable. For instance switch values of type SNVT_switch are scaled and displayed as a numeric pair describing the switch level and its corresponding binary state (see Section 2.4.5 for an explanation of SNVTs). However, this presents a problem when a user-defined variable not covered by a standard type is used. The data is presented as a pure binary field, which has to be interpreted manually. This basic type of monitoring tool is shown in Figure 4-16.A.

Data presented in this manner has a number of disadvantages. The abstract fields of user defined data are not easily understood. Update intervals are frequently too slow to catch rapidly changing events. Moreover, high frequency data is extremely difficult to analyse when presented in a textual form. Relationships between the multivariate fields are also hard to see.

For these reasons a more sophisticated control program was developed, this can be seen in Figure 4-16.B. Values from specific network variables that require monitoring are presented on the display adjacent to a pictogram of their respective leg. In general terms, this type of control program is typical of most fieldbus user interfaces, examples can be seen in IEC98 and ECHE98b. For the ELMA specific control program shown the network variables made available by the middle level functional blocks of the performance model can be monitored and altered.

The final evolution of a user interface developed for the hexapod was to allow extraction of the full performance and localisation model of each leg, this can be seen in Figure 4-16.C. The preceding software control program (Figure 4-16.B), only allowed a specific set of network variables to be monitored. Whilst these variables are part of the node's Control Object Model they are not obtained by examining it. Instead, since the monitoring program is custom written and will only ever communicate with a robot network, they are specified at the time the program is written. However, in the final case the node's full performance model, and consequently the functional blocks that compose it, is downloaded to an application viewer. The viewing program then instantiates appropriate objects (C++ or JAVA based constructs) which automatically configure themselves to display appropriate information from the robot.

The example shown in Figure 4-16.C is obtained by taking the head node localisation model (previously shown in Figure 4-15) and combining it with the localisation model from a particular leg node. The body of the robot stays static at all times. However, the leg node localisation model is drawn at a position based upon the current value of the leg's output network variables. This manipulation of the graphical localisation model is performed by the related performance model executing on the PC based host. Additionally, a grid of cells can be seen that correspond to the visit counts generated when using the reinforcement learning algorithm from Section 4.4.

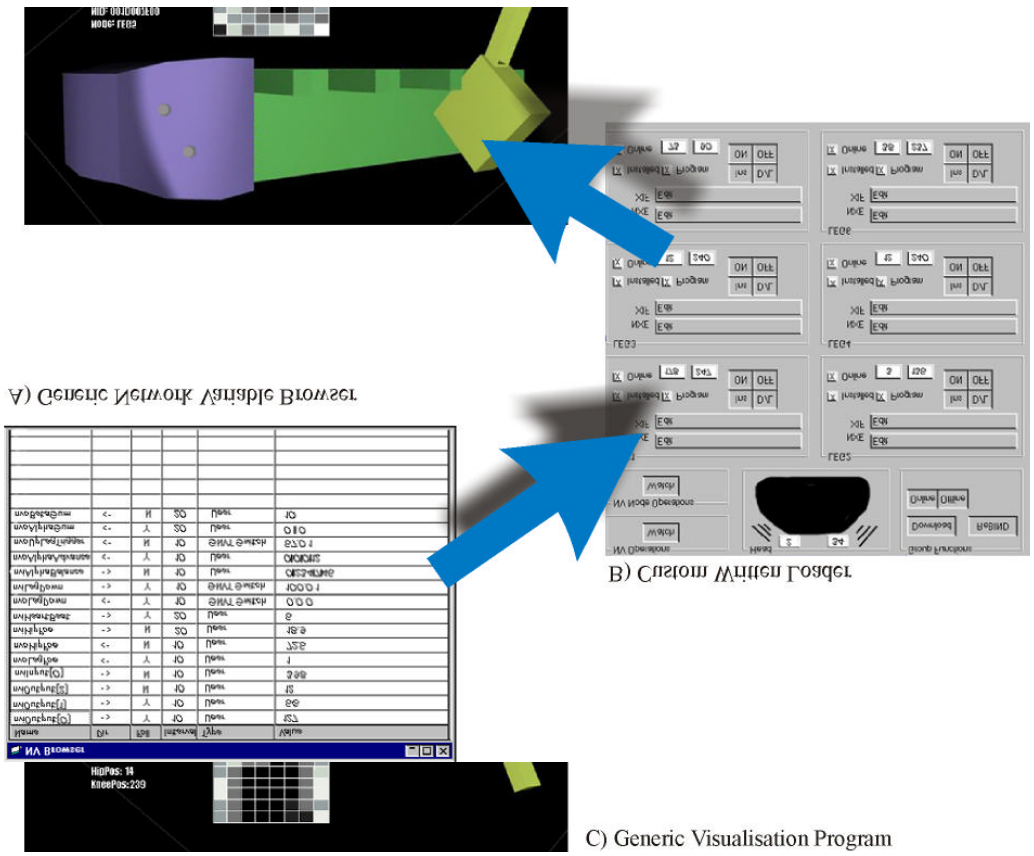

Figure 4-16 Progression of Visualisation Programs

It can be seen from this example of a sophisticated fieldbus monitoring program that a number of interaction and visualisation methods are possible. The generation of dynamic user interfaces by extracting modelling information from the node itself in this way is believed to be unique. Chapters 5 and 6 go on to study the implications of this and to explain in detail the operation of the viewing program that allows this software to function. Considering the custom written loader, for a simple network of nodes a dedicated user interface is simple to write and normally offers adequate performance. However, when the user must interact with a large distributed control system then efficient methods of visualisation and interaction become necessary. These needs are addressed by the Damocles software in Chapter 6. During the course of this research it became apparent that for large networks where the user is surrounded by many nodes a novel interaction method was needed that permitted simple examination and control of the networked nodes. For this reason, augmented reality was selected as a suitable candidate for the implementation of an immersive virtual control system; this is described in the next Chapter.

4.8 Conclusions

In this Chapter, a variety of different software architectures have been examined. These have all been implemented on the ELMA hexapod. The chosen architectures represent a structured and methodical approach to the construction of distributed control software. In particular they show that complex and sophisticated software can be implemented on relatively low performance hardware when that hardware offers certain intrinsic networked capabilities.

In Chapter 1 two statements regarding the implementation of distributed control systems were made. That is, low performance yet cost effective systems are often overlooked by academia when implementing control architectures. Secondly, complex control algorithms are often ignored by industry because it is perceived that they cannot be implemented on the chosen processors. The implementation of a subsumption architecture and reinforcement learning on a network of Neuron processors demonstrate that both of these statements can be negated when carefully constructed software is used.

Both the Reactive (AFSM) controller and the Q-Learning leg trajectory controller have been compared to the more simplistic centralised control scheme presented in Chapter 3. Experimental data indicates that localised, distributed, processing can be used to reduce network bandwidth usage with minimal effect on software complexity. Furthermore, comparison with other published material indicates that these solutions are not significantly worse than any other and in some cases (for instance end effector Q-Learning) offer better performance.

The success of the ELMA hexapod, in terms of reliability and demonstrability, show that *LonWorks*® is a viable platform for developing sophisticated multiprocessor based robotic systems.

It has been shown that a performance model composed of a number of functional blocks can be used to aid the design, development and testing of this robot system. Furthermore, the performance model has been shown useful in the visualisation of the robot where it is used to group software blocks into readily observable units. These units offer the user custom interfaces specific to the device in question giving him relevant data in the most appropriate format.

The Performance Model is part of a much greater whole called the Local Object Model. This concept, comprised of a number of different classes, is useful when maintaining, monitoring and interacting with distributed control systems. For the monitoring of ELMA the localisation model has been used to store graphical data corresponding to the body and legs of the robot. In this case, the performance model operates upon the localisation model to provide a dynamic display of leg orientation and related data. The software binding these items together is examined in the next two Chapters.

With the installation of a large scale distributed control system in the Department of Cybernetics it became apparent that the ideas developed on ELMA could be enhanced to cater for environments where the user is enveloped by the control system. In this case, many disparate nodes are placed around the environment. The user requires the ability to operate with either a subset of devices in his locale or all of the devices of a certain type. Since many nodes in the built environment are not observable (being built into ceiling spaces and so on) then a visualisation tool must be able to display spatial context relevant information. In other words, a node or virtual representation of it should appear in the users workspace at a position consistent with its world location to aid recognition and interaction.

In the next Chapter the development of an augmented reality based visualisation tool is shown. In particular a number of hardware and software related issues are identified as governing the performance of such a system and these are addressed. Based upon this background Chapter 6 goes on to detail the operation of a fully immersive interaction tool for use with distributed control systems. This same tool has been used as the *viewer* application mentioned in Section 4.7.3 thus demonstrating its applicability to a wide range of monitoring and control tasks on fieldbus systems.