DIVISION OF COMPUTER SCIENCE

Modelling Changes in Organisations for a Complex World

P N Taylor
E De Maria

Technical Report No.228

September 1995

Modelling Changes in Organisations for a Complex World

† P. N. Taylor and ‡ E. De Maria.

† Division of Computer Science and ‡ Business School,
University of Hertfordshire, College Lane, Hatfield, Herts.
AL10 9AB. U.K.

Tel: +44 (0) 1707 284763
email: comrpnt@herts.ac.uk

[1] In this paper we discuss how to model the dynamic restructuring of a complex organisation and its information flows to ensure that the organisation continues to function despite fundamental changes taking place. Changes occur mainly because of functional and environmental effects.

Decisions can be made rapidly and effectively if key information about each level of the organisation is stored in an efficient and accessible way. We show how to model the organisational hierarchy such that it can be adapted quickly and easily to cope with new challenges.

As an example, we describe how this model is used in a wargame which simulates two military organisations in conflict. The model itself has far wider applications since it can be used to monitor the flow of command decisions, logistics and changes in unit position and status.

We introduce an object-oriented computer model which views an organisation as a set of interacting objects; mathematical process algebras are used to model these interactions.

[1] This work is funded by sponsorship from British Aerospace Defence Co. Ltd. (Dynamics Division), Six Hills Way, Stevenage, Herts. SG1 2DA. U.K. Principal BAe Project Engineer, J.L. Young.

Introduction

The relationship between Operational Research and the military has had a long and distinguished history, dating back to just before the second world war. The success of the work carried out in those pioneering days of O.R has assured it a permanent place in association with the military.

Many of the mathematical techniques that we use today had, at that time, not been invented. Neither had the powerful computers that we now use to carry out such tasks. Despite the lapse of time since those early days of O.R the emphasis of military O.R is still towards applying available tools and methods to study military operations. Carrying out this military work we find that private companies, individuals and universities are the main suppliers of material in this area [3]. The application area with most entries (taken from a study between May 1984 and May 1991 [3]) were warfare and manpower. Our own particular study concentrates upon the communication of information in warfare and the computer simulation of combat between two battalion-sized forces.

Presently, we entrust many of the calculations and assessments that are carried out on behalf of O.R to mathematical computer models. These models provide us with large volumes of data about the system under review so that representative studies of a problem area can be undertaken. Mathematical models are an essential part of military O.R studies since large scale military exercises are beyond the scope of all but government-based organisations, mainly due to their cost.

We can break our study into 3 areas: (i) reality, which is where we reside, (ii) a model of reality. For this particular study perhaps a manual wargame. Finally, (iii) a software interpretation of the model of reality. This paper concentrates upon this third area, the software interpretation.

The computer model that we introduce in this paper uses advanced software modelling techniques that have evolved within the field of computer science over the past few years; techniques centred around object-oriented design methodologies [6,15] and object-oriented programming languages [11,16]. These new software practices give us a more flexible approach to solving problems than was possible in the past.

We apply our software model to the dynamic organisation of a battalion-sized military combat force and draw upon a subset of problems encountered by all organisational structures in either military, industrial or commercial environments; namely, coping with changes to the communication structure of an organisation which are influenced by either a potentially hostile environment or changes in internal functionality. Our model is an abstraction of military command as we only capture units from battalion level downwards to individual squads of men. Consequently, we only model six levels of the overall military command structure.

Certain assumptions are made regarding our software model. We simplify the communications model normally adopted by the military by insisting that units at level n of the command structure can only communicate with neighbouring units at levels $n-1$ and $n+1$. In our computer model destruction of the force commander's Battalion node leads to cessation of the wargame. In reality we recognise that numerous levels of redundancy are introduced to ensure survival of the force and its command structure. Units resident in our software model may not communicate with other classes other than

those listed in their attributes list. At this early stage of development our work draws upon theoretical ideas regarding data flow and the rerouting of communications links.

The organisation of this paper is as follows. Section 1 introduces the problem area by way of our computer simulation model. We address a central issue of maintaining the flow of information despite disruptions to the combat communications network. Section 2 discusses the dissemination of command that so typifies the management structure of a military organisation and other civilian organisations which are also engaged in life threatening situations (e.g: the emergency services). Section 3 introduces the relevant storage of information in terms of efficient access, communication and alternate sources of information. Section 4 presents a strategy for adapting information storage depending upon environmental and functional changes to the network. Section 5 discusses the locality of information and effective and efficient data transfer (i.e: getting the relevant data to where it is needed). Section 6 discusses a multi-layered solution using object-oriented software design methods and implementation. Section 7 suggests ways of identifying errors within elements of the organisation before they can become a threat to the entire network; namely a strategy for minimising critical failures. Section 8 discusses the stability of the proposed software data structures. Section 9 formalises the use of objects and the dynamic connections between them using the π-calculus [8] to show how connections can be substituted between nodes when failure occurs. Section 10 draws conclusions from the work carried out so far and suggests areas of further research and development to improve and enhance our proposed model.

1 Disruption of Information Flow

Military organisations the world over adopt a similar hierarchical view of management. Policy and strategy is passed down from senior ranks to lower echelons to be implemented. We use a similar hierarchical view of management to model the command structure behind our battalion level wargame; entitled EMBLEM (Empirical Man-in-the-loop Battalion Level Effectiveness Model).

The primary goal of EMBLEM is to provide British Aerospace (Dynamics) with weapons related information in a man-in-the-loop combat simulation environment. EMBLEM is intended to provide a platform from which statistics regarding the use of both direct fire and indirect fire weapons can be gathered. We use the computer simulation of engagements between opposing forces to generate simulated results of weapon systems performance. The computer model enables us to produce realistic data regarding the effective performance of particular weapon systems as well as highlighting areas of concern related to their design and deployment.

In our computerised wargame two commanders 'do battle' by positioning fighting units upon a map displayed upon the computer screen and then issue orders to those units in order for them to reach their assigned objectives. Underlying the map a terrain database (GIS) will supply height information for line-of-sight and movement calculations to be performed so that the possibility of visual engagement can be determined. Flight characteristics for different weapon systems will be fed into the

simulation enabling accurate fly-outs to be recorded.

EMBLEM is a *closed* wargame; each player does not have full knowledge of the position or strength of the enemy. Separate terminals are used to display each player's own force positions and any known enemy positions. Intelligence gathering sensors will relay enemy details to the commander of each battalion (i.e: each player). Commands are issued at the terminal, directed towards individual units or groups of units on the battlefield.

A senior player takes the role of neutral umpire, determining the validity of each move and request from each of the two players. Historically, the Red team are the attacking force, with the Blue team defending key installations or primary positions on the battlefield. A typical engagement area for our model is 50 kilometres^2, an area dictated by the weapon systems that we intend to study. Larger, more complex combat scenarios have been modelled for different studies with different objectives for the system designers [2].

The data structure used within EMBLEM resembles a tree structure where one complete tree models each fighting force. Object-oriented software construction techniques [6,15] are used to provide encapsulated storage for each node on the tree. Note that we use the terms *node, object, entity, unit* and *process* interchangeably to describe the same element in our model. At the software level each node is an object containing local state information, behaviour and links to subordinate nodes and its own commander. We use a dynamic table (stored as a file) to maintain the actual location of each subordinate node.

Presently, messages take the form of method calls to specific objects which are akin to requesting an object to perform a certain task rather than passing a packet of information containing a task to each object. Future development intends to pass messages as parameters to method calls to increase the flexibility of the software model.

Although unlimited subordinate nodes can exist at each layer in our software model a battalion-sized force is usually restricted to the units shown in Figure 1.1.

```
    1 x Battalion
    6 x Companies
   24 x Platoons (4 per Company)
   96 x Sections (4 per Platoon)
  480 x Vehicles (5 per Section)
  480 x Squad (1 per Vehicle)

Total of   1087 units per force.
```

Figure 1.1

To model conflict between the Red and Blue teams we construct two tree structures. Communications propagate through the six layers of the tree structure in order to task each object with a specific mission. Only the last two layers of the structure represent active units on the battlefield. The first four layers simply allow us to capture the chain of command. Orders filter down to vehicles and squads from the entities above. Figure 1.2 shows one battalion structure partially populated with objects, one sub-branch of the whole structure is presented to aid clarity. For example, a typical fully

populated diagram of a battalion-sized force would require another 20 platoons, 92 sections, 475 vehicles and 479 squads. Based upon the overall size of each force the scope of the entire communications network becomes apparent, hence the need to sub-divide communications at each command level to maintain simplicity.

The Battalion node is head of the organisation and from each player's perspective is used as a point of entry into the structure. An order issued at the Battalion node is disseminated through the chain of command to the appropriate vehicle or squad.

In EMBLEM, as in reality, communications flow in both directions through each layer in the structure. Each entity connects to n subordinates as well as its own commander. Therefore, any node can identify its commander and who it itself commands.

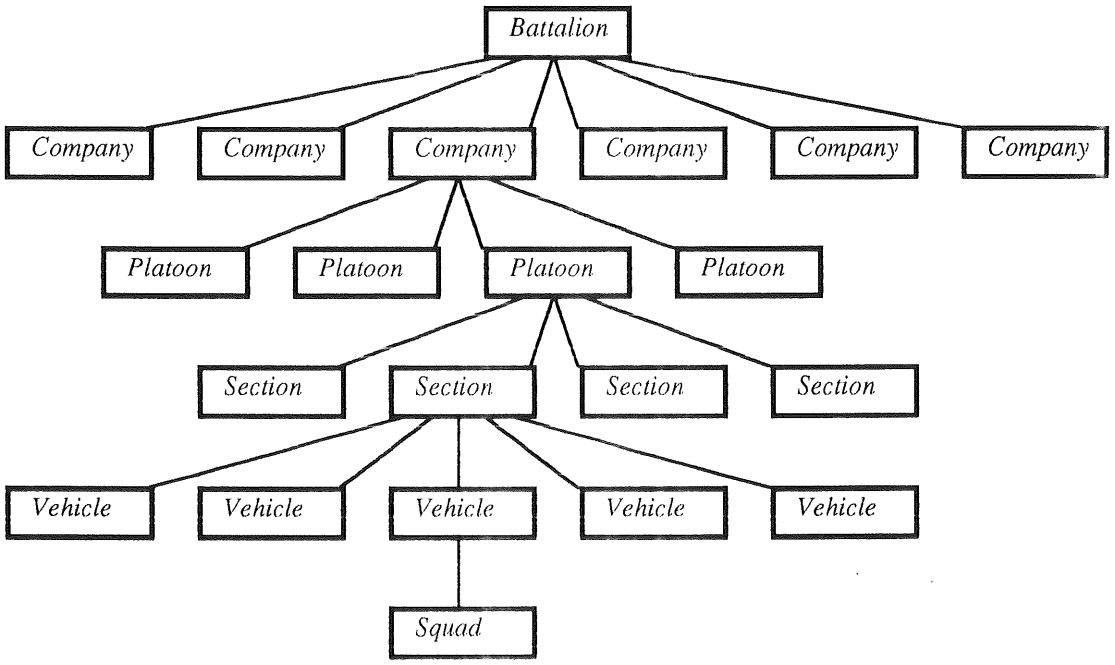

Figure 1.2

It is paramount for any military unit to continue to function despite serious disruptions to its information pipelines and command structure. At this stage in its development much of our theoretical work and software implementation parallels that of [1], although we simplify the fine details of logistics and presently omit electronic countermeasures. Whereas [1] concentrates upon a complete model of an operational unit this paper focuses upon certain key issues of the complete system; namely information and flow and communications between battlefield units.

Throughout the remainder of this paper we attempt to show how such a structure can be maintained so that key information continues to flow around the system regardless of the loss of a certain number of nodes or, perhaps, entire levels of the original structure.

2 Dissemination of Command

One of the key issues surrounding the survival of an organisation in a life threatening environment is that of its ability to adapt to change; akin to evolution in biology where only the fittest and most flexible survive. In the context of military operations we regard a life threatening environment as one which is continually changing, cannot be predicted and seeks to influence the organisation in some way by attempting to compromise elements of the structure.

Damage to communications within an organisation can be minimised by adopting a strategy of disseminating command. In a military structure this entails a certain amount of autonomy at each of the six layers that we have identified. Overall control is still maintained at the head of the structure and overrides any locally issued commands. In the absence of communications from a higher authority each node should take control of itself and communicate with its subordinates in the hope that communication with managing nodes can be restored.

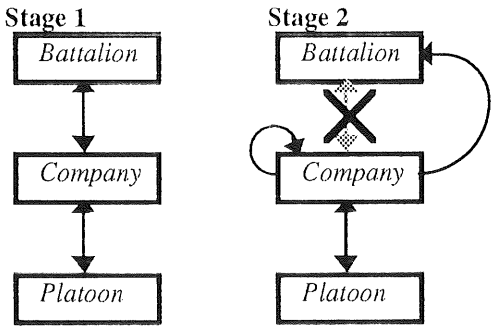

Figure 2.1

Figure 2.1 (Stage 1) illustrates normal two-way operational communications. Stage 2 shows the failure of a link between the battalion commander and one of his company commanders. Subsequent communications from the battalion are lost and therefore the company assumes control of itself, maintaining links with subordinate nodes whilst sending messages back up the hierarchy to its commander regardless of any further commands it may or may not receive.

One important implementation issue is the collection and storage of relevant information to enable each node to carry on without directions from its superiors. One software solution is to share the location of each node's commander and its subordinates with all other nodes on the same level of the data structure. Problems associated with this approach include the administration of information should some nodes cease to function. The integrity of the information across the network would be lost if there is a break in command. Also, implementation constraints are imposed by storage (memory) and speed of execution (hardware). With 2174 nodes (representing fully populated forces for both Red and Blue), each requiring up to 40 bytes of data storage we soon encounter large overheads in terms of both access times and memory usage. For any computerised combat model to be effective response times are critical to ensure feedback to the commanders so that decisions can be made regarding commands issued to respective forces.

Alternative strategies for information storage could include the creation of a *God* node which holds information about every other node in the entire network. Arguments against this particular form of network management are that failure of the *God* node will guarantee system failure; similar in context to destroying the command structure by targeting the senior commander. Within complex management organisations efficiency must be traded against effectiveness. Single, isolated stores of information are not recommended as the network is under severe threat should the primary store be compromised, especially if there is no information backup storage facility within the system.

Only information which is necessary for the survival of each node should be stored locally. A question arises as to the identification of information that is regarded as necessary (the reader is referred to [9] for more discussion on this topic). In our software model we concentrate upon maintaining the links between nodes in the data structure by using existing nodes to provide a bridge between one layer and another. This link information is stored locally at each node in the form of a single value (the unit's commander) and a table of values (the unit's subordinates).

A unit can use these locations to setup an alternate route to deliver required information. We use a dynamic list of destinations that each node can reference in order to send commands to specific units or all units under its command. Blanket commands that affect all lower echelons are simple to implement but specific commands incur a higher overhead as the unit in question needs to be sought out and then tasked. A 'hit' on the specific unit may only occur after a number of 'misses', where the communications were routed through incorrect branches of the tree. A key issue arises from the software implementation of routing messages with specific routing information embedded within each packet. Increasing the message size with such information incurs a heavier load on the communications network as all messages are increased in size. Efficiency is traded off against the number of 'hit's and 'misses' of each packet (see [12] for more information regarding routing algorithms and efficiency).

3 Information Storage

We have already argued that the notion of storing a lot of network structure information per node is inefficient and causes depreciation of performance within the system. The question remains "what information is required by each node to aid its survival and to allow it to perform efficiently?". Similar issues are addressed in [9] regarding the placement of information in distributed systems to aid efficiency whilst maintaining data integrity. Although we regard Battalion, Company, Platoon and Section nodes as simply transmitters for receiving and forwarding information they do have some physical presence on the battlefield. Each one of these four entities represents a command post which can be subjected to attack by the enemy. Should this happen then command would have to be transferred to alternate sites to enable the force to operate in a coordinated manner.

In our EMBLEM model an end-of-game signal occurs when the Battalion node is put out of action. There is only ever one Battalion node per combat force. The two physical entities that we are most concerned with are vehicles and squads. Strict rules govern the use of vehicles and squads when

receiving and acting upon commands. Vehicles can pass commands to squads whereas squads are terminal nodes and do not pass commands onward. Only 'live' vehicle crews and squads can receive commands. A vehicle without a 'live' crew or squad in attendance is inert.

From the perspective of our software model, if a node should cease to play any further part in the wargame (either through failure or actual destruction) then its local state information is lost. However, we do not lose the information about the dead unit's subordinates. Consider the example in Figure 3.1.

Company c1 communicates with three platoons. Messages come down from battalion b1, through company c1 and on to the platoons p11, p12, p13 and so on down the chain of command. The failure of company c1 means that communications from battalion b1 will be lost if some other communications bridge is not set up to act as a replacement.

Figure 3.1

In our software model each node maintains a list of subordinates with which it communicates. The list itself is in the form of a file which contains the memory addresses of each subordinate node. An analogy of the list would be radio frequencies and code words with each subordinate communicating on a slightly different frequency. Any node taking on the responsibility of a failed node would simply add the new addresses to the end of their list. When that node gets its communication slot it simply passes each message to each node pointed to by the addresses in its file list. To illustrate failure recovery in our particular model consider the course of events shown in Figure 3.2.

Company c2 assumes command from company c1. Company c2 adds platoons p11, p12, p13 to its existing list of platoons (p21, p22, p23). Communications are then restored and the structure is once again fully operational.

Figure 3.2

We could automate the selection of an alternate bridging unit to carry the messages of a failed unit. However, as EMBLEM is a man-in-the-loop simulation we offer alternate bridges to the player who may then choose an appropriate unit to take control. Figures 3.3.1 and 3.3.2 further illustrate the previous discourse from Figures 3.1 and 3.2.

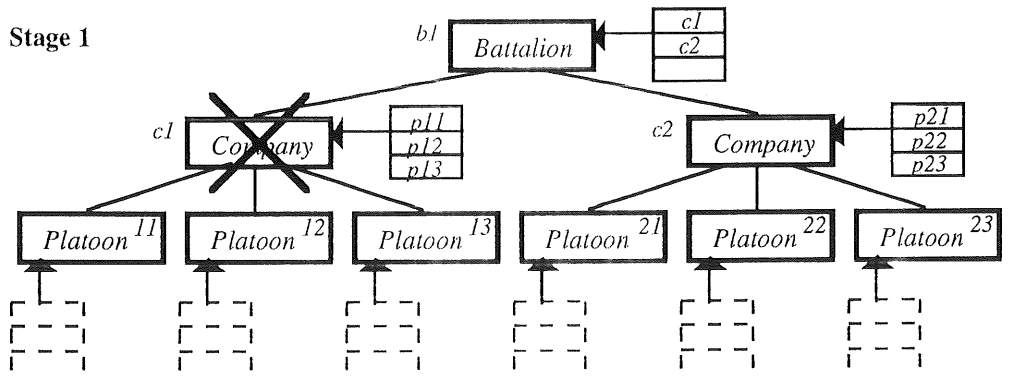

Figure 3.3.1

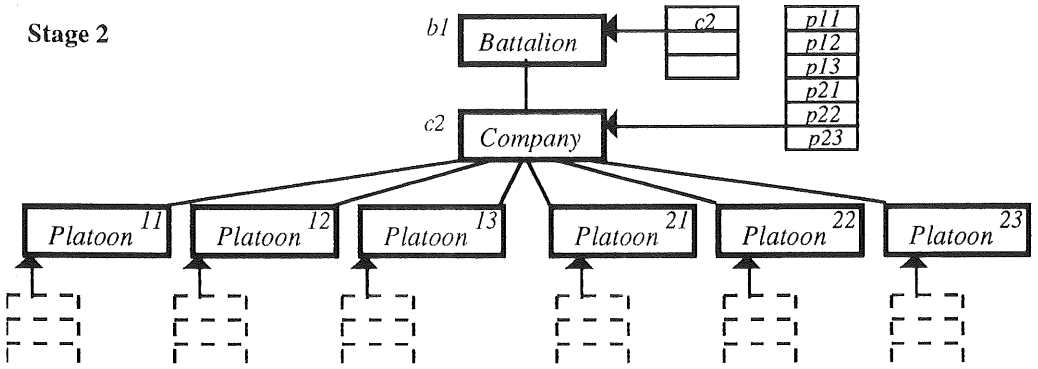

Figure 3.3.2

Figure 3.4 shows a source code extract for the sending of a message from a platoon to a series of sections, up to some maximum constant value `kMaxSECTION`. A collection of these constants form systems parameters that govern how many units of a particular class can be contacted via each other class in the system. The boundary set by these constants is known as the critical failure threshold which restricts rerouting and backup unit allocation (see section 8 for more detailed discussion on backup units). For the EMBLEM project we use the object-oriented programming language C++ [11], hence both platoons and sections are modelled using class constructs.

```
Boolean PLATOON::SendMsg(MSG command)
{
   FILE *filePtr;
   short count = 1;

   if((filePtr = fopen(this->getPLLinkFileName(),"r")) == NULL)
         return(false);

   while(count <= gMaxSECTION && (fscanf(filePtr,"%p",&ptrSECT) != EOF))
   {
         ptrSECT->SendMsg(command);
         count++;
   }
   return(true);
}//end-method-PLATOON::SendMsg
```

Figure 3.4

In our software model we assume a maximum permissible number of node failures. Once exceeded further rerouting and backup node allocation is not possible. Eventually an end-of-game signal occurs as too many units have been destroyed or put out of action. One system global variable that we use is that of a maximum value for subordinate nodes. A limit can be placed upon the number of subordinate nodes that a commander can communicate with. This value represents the number of entries in the link table associated with each node.

If all communications fail in the network then it is intended that each unit will proceed with their last command and halt until further instructions are forthcoming. Presently, we do not model any

form of intelligence in the units on the battlefield.

In section 6 we introduce a more advanced method of redundancy which builds upon the ideas presented here. Our advanced method adds another level of redundancy into our model by introducing extra nodes into the system to replace failed nodes rather than extending the coverage offered by existing nodes.

4 Adaption of Information

It is logical to assume that an organisation's environment will influence the behaviour of that organisation. In EMBLEM the orders that are issued by each commander to their respective forces are based upon experience and available intelligence. In reality this intelligence information is gathered from sensors on the battlefield, either manned or unmanned. In our software model intelligence information is presented in the form of console messages which are logged in a file for on-going reference and post-game analysis.

We regard environmental effects as those primarily caused by enemy units. Internal (a.k.a functional) effects are caused by node failure or internal processing carried out by autonomous nodes which are only indirectly effected by the enemy

Each type of influence (either internal or external) can cause the links within the communications network to change. We do not consider that the actual organisational structure will change, only that the nature of the connections within the organisation will change. Connections between units must be reviewed in order to maintain successful communications across all six levels in the chain of command.

In our software model elements of the command structure can be tested for failure by evaluating the state of each node, as shown in the diagrams in Figure 4.1.

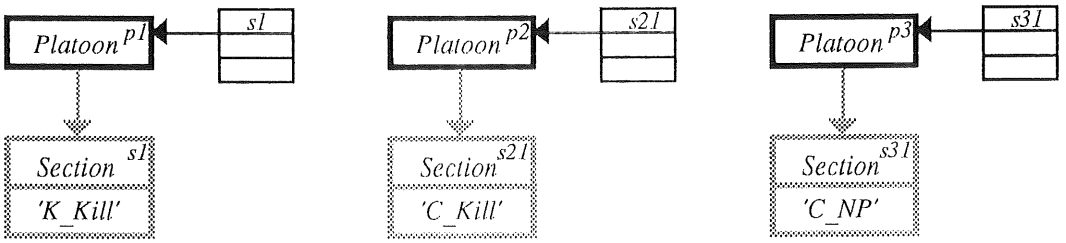

Figure 4.1

Section *s1* in Figure 4.1 is not in a fit state to carry out any orders as it is 'dead'. The state *K_Kill* effectively removes the unit from the battlefield as it can play no further part in the simulation. In our software model, if a unit forms a link to lower echelons then it must be replaced or bridged, otherwise all subordinate nodes from the dead node onwards will be not be able to be contacted by the force commander. In Figure 4.1, section *s21* has suffered a 'communications kill' (*C_Kill*) and cannot

be communicated with or communicate with any other unit. At some later stage in the battle we may signal adequate repairs to the communications equipment aboard section *s21*, thus enabling it to rejoin the communications network. Section *s31* is temporarily unable to receive communications (*C_NP* - communications not possible). This may be due to electronic jamming or even terrain shielding. It is only a temporary situation but any messages sent during the period when *s31* is in this state will not be received.

In an aggressive environment the creation of *dead zones* within a organisation's communication network is expected as soon as nodes start to fail (i.e: units being compromised on the battlefield). The structure of our software model draws upon the experience of military computer networks which have the ability to survive in hostile environments [12]. These types of network can reroute information to ensure survival. With our computer model we seek the same results, accepting failure of the entire network only when a critical number of key nodes are destroyed (i.e: surpassing the critical failure threshold). The particular strategy that we intend to follow is that of flooding the network with messages [12]. Due to the relatively small size of our network in terms of depth (i.e: 6 layers) we do not anticipate unacceptably large overheads as messages that 'miss' their target will not travel beyond *6-n* layers before reaching a terminal node, where *n* is the layer issuing the message. Figure 4.2 shows an example scenario featuring 'hit' and 'miss' nodes.

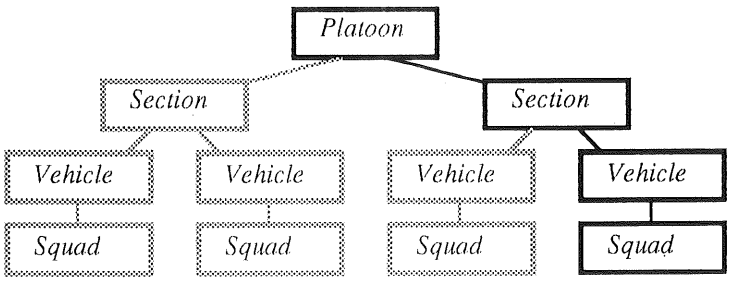

Figure 4.2

The shaded units represent those leading to a 'miss'. The squad destined for the command is shown at the end of the bold path.

4.1 Object-Oriented Design of EMBLEM Data Structures

Each layer in our EMBLEM data structure represents a separate class of object, where an object further represents a combat unit in a battalion-sized force. The classes that we define for each layer are listed in the table in Figure 4.1.1. The [2] DIS object at the top of the table in Figure 4.1.1 does not appear in our conceptual command structure (as shown in Figure 1.2) as it is considered to be an abstract class and consequently has no physical representation within our model.

[2] Note that DIS (taken from the IEEE Standard 1278 for Distributed Interactive Simulation [5]) is our own interpretation of the recognised military simulation standard used to connect several different types of simulation into one virtual environment.

Layer	Class Name	Unit Name
0	DIS	Global Attributes per class
1	BN	Battalion
2	COY	Company
3	PLATOON	Platoon
4	SECTION	Section
5	VEHICLE	Vehicle
6	SQUAD	Squad

Figure 4.1.1

The structure of the DIS class is derived from those attributes common to all other classes in our software model. The DIS class can be regarded as the grandparent object. The remaining six objects are then derived from the DIS class. Each object class (BN,COY,...,SQUAD) inherits its basic behaviour and state information from the DIS class and then extends that information to include details about its immediate neighbours and other specialist attributes. These specialisations make the object an individual class from which other classes could be derived.

Object-oriented software construction allows us to reuse the structure of the DIS object, minimising duplication within the source code. The DIS class is an abstract class template which provides a given form for its derived classes. These in turn are also abstract class templates. It is only during instantiation of either of the layers 1 through 6 that an instance of an object actually exists in a form that we can manipulate and command. Figures 4.1.2 to 4.1.8 show the C++ source code extracts defining each class of object, from level 0 (zero) DIS to level 6, SQUAD.

```
class DIS
{
  private:
    Force        forceID;
    char         entityName[kMaxNameSize];
    float        entityLocation[3];
    float        entityVelocity[3];
    float        entityOrientation[3];
    FightingState state;

  protected:
    ...class methods defined here
}
```

Figure 4.1.2

```
class BN : private DIS
{
  private:
    COY          *ptrCOY;

  public:
    ...class methods defined here
}
```

Figure 4.1.3

```
class COY : private DIS
{
  private:
    BN           *ptrBN;
    PLATOON      *ptrPL;

  public:
    ...class methods defined here
}
```

Figure 4.1.4

```
class PLATOON : private DIS
{
  private:
    COY          *ptrCOY;
    SECTION      *ptrSECT;

  public:
    ...class methods defined here
}
```

Figure 4.1.5

```
class SECTION : private DIS
{
  private:
    PLATOON    *ptrPL;
    VEHICLE    *ptrVEH;

  public:
    ...class methods defined here

}
```

Figure 4.1.6

Note that the class VEHICLE in Figure 4.1.7 has two *FightingState* variables. These hold the current state of both the vehicle and the crew. Commands are carried out provided that the crew is in a position to receive them (not *K_Kill*, *C_Kill* or *C_NP*) and that the command is reasonable and can be carried out (e.g: movement command and *vehicleState* ≠ *M_Kill*).

```
class VEHICLE : private DIS
{
  private:
    SECTION            *ptrSECTION;
    SQUAD              *ptrSQUAD;
    short              numberOfCrew;
    WEAPONSTRUCT       vehicleWeapons[kMaxWeaponSize];
    FightingState      vehicleState, crewState;

  public:
    ...class methods defined here
}
```

Figure 4.1.7

```
class SQUAD : private DIS
{
  private:
    VEHICLE            *ptrVEHICLE;
    short              numberOfMen;
    WEAPONSTRUCT       squadWeapons[kMaxWeaponSize];

  public:
    ...class methods defined here
}
```

Figure 4.1.8

The creation of each object of a specific class (its instantiation) is brought about using the C++ function *new()*, which allocates storage for the object and assigns a pointer variable with the address of that object. Every method (a.k.a operation or function) that we request of a particular instance of a class uses the assigned pointer variable. Figure 4.1.9 illustrates object creation for each of the classes in our model, together with example method calls using the pointer variables.

```
BN          *ptrBN = new(BN);
COY         *ptrCOY = new(COY);
PLATOON     *ptrPLATOON = new(PLATOON);
SECTION     *ptrSECTION = new(SECTION);
VEHICLE     *ptrVEHICLE = new(VEHICLE);
SQUAD       *ptrSQUAD = new(SQUAD);

ptrBN->moveBN(newPosition);
ptrCOY->linkCOY_BN(ptrBN);
ptrPLATOON->setPlatoonForce(Blue);
ptrSECTION->printSection();
ptrVEHICLE->getVehicleState();
ptrSQUAD->armSquad(weapon);
```

Figure 4.1.9

In C++ the arrow notation (->) is used to request a specific method pertaining to a class. In Figure 4.1.9 the methods invoked can be seen after the arrow. Parameters required by each method appear in the braces following the method name.

5 Locality of Information

By following the pointers housed in each object a unit knows the identity of its commander and also the identity of those units that it commands. Units do not hold information about other units of the same class (i.e: units on the same level of the command hierarchy). Formally, nodes at level n do not store information about other nodes at level n, only information regarding a single n-1 level node (its commander) and *gMax<ClassName>* multiplied by n+1 level nodes (its subordinates).

As each level of the EMBLEM data structure is represented by a distinct class of object we could consider a modification to the failure recovery method initially introduced in section 3, where an existing node takes up the responsibilities for communications on behalf of some failed node.

The proposed strategy for modification uses a backup supply of nodes, initially unused, that are available to be called upon should units start to fail. A global variable governs the total number of each class of object available as a backup. However, certain restrictions apply. For example, only one instance of class BN is permitted and no further instances of classes VEHICLE or SQUAD are allowed. These last two classes represent physical entities and, as such, we cannot consider replacing them should they fail; once you're dead you're dead! The global variables *gMaxBackup<ClassName>* store the number of backup nodes of each class available to the network at a time of crisis.

The C++ source code extract in Figure 5.1 helps to crystallise the idea of backup objects being used to support successful communications.

It is worth pointing out at this stage that the original strategy for using existing nodes to aid communication still exists and comes into effect once all backup nodes have been exhausted. The modification proposed in this section simply provides yet another level of redundancy to our existing

command structure, further enhancing its resilience against enemy attack.

```
while((count <= gMaxCOYLimit) && (fscanf(filePtr,"%p",&ptrCOY) != EOF))
{
   state = ptrCOY->getCOYState();
   if((state != K_Kill && state != C_Kill && state != C_NP) &&
         gMaxBackupCOY > 0)
   {     //if spare node available then allocate and use failed node's links
         newCOYPtr = new(COY);
         newCOYPtr->SetLinkFile(ptrCOY->getLinkFile());
         newCOYPtr = ptrCOY;
         delete(ptrCOY);
   }     //...else if no spare nodes available then use existing node...
   else if((state != K_Kill && state != C_Kill && state != C_NP) &&
               gMaxBackupCOY <= 0)
   {
         tempCOYPtr = ptrCOY->getNextCOYPtr();
         tempCOYPtr->LinkFiles(tempCOYPtr->getLinkFile(),
                                 ptrCOY->getLinkFile());
         tempCOYPtr->ResolveLinks();
         delete(ptrCOY);
   }     //...else simply issue the command
   else
         ptrCOY->SendMsg(command);

   count++;
}//end-while
```

Figure 5.1

The extra step of using spare resources to help maintain communications throughout the network can be further illustrated in Figures 5.2 and 5.3.

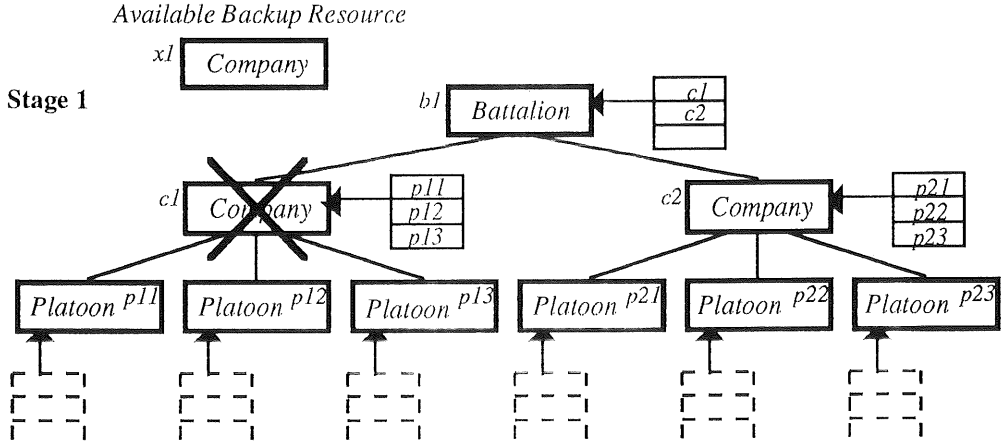

Figure 5.2

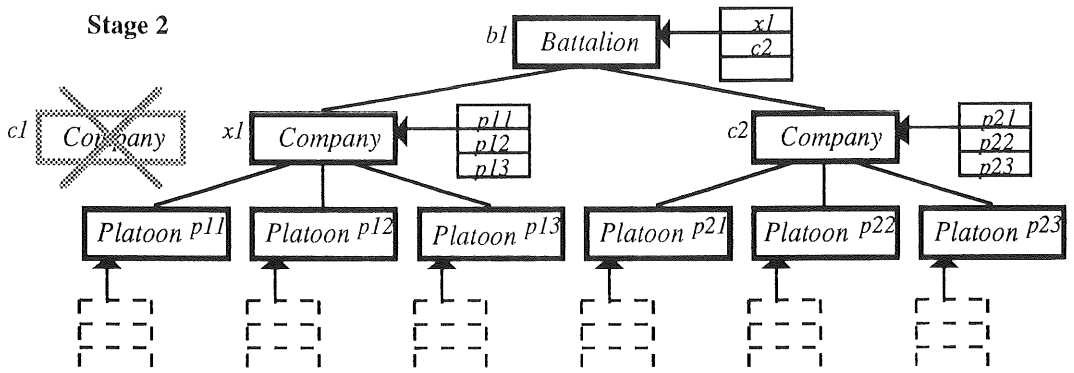

Figure 5.3

The node for deletion is shown as the shaded Company object *c1* in Figure 5.3. Note the updated table for the Battalion *b1* node and the use of the original *c1* Company list table with the new Company node *x1*. When modelling an organisation with a poor logistical capability we reduce the number of available backup nodes. Difficult terrain and hazardous resupply routes will also have an effect on the total number of excess units at an organisations disposal. The umpire for each game is responsible for determining the bounds for backup objects and maximum communications for each class.

Within our data network the rate of success in a hostile environment remains high as key nodes can be substituted if they fail. Memory considerations associated with extra node allocation cannot not be ignored but failed nodes will release their allocated storage back into the pool of available memory so the reuse of address space is therefore possible.

6 Multi-layered Management Solution

For the sake of efficiency it seems logical to service any request locally rather than implicitly request other distant nodes to undertake the task. In the software model used to build EMBLEM local help comes in the form of object classes that can aid successful communications.

In object-oriented programming certain methods (a.k.a functions) are associated with each object type. Common actions are dealt with at the top of the object inheritance hierarchy in the DIS object (see section 4.1, Figure 4.1.1). Note that the inheritance hierarchy is a different concept to that of the command hierarchy. Commands that are issued to assign objects to fighting forces (i.e: Blue or Red), issue meaningful names (such as Platoon Bravo, Delta Section, 2nd Battalion), move objects around the battlefield and set an object's fighting state are all delegated up to the original methods associated with the grandparent class DIS. Commands such as assigning a crew, arming a vehicle and engaging the enemy are dealt with locally due to the uniqueness of the action.

Communications and orders propagate through the chain of command in a single direction. In some cases local action is required prior to the command being forwarded to the next level of the

command structure. Intelligence and requested information is passed back up the chain of command when necessary. Due to the nature of the branches in our data structure (see section 1, Figure 1.2) information from the battlefield reaches the top of the command structure with minimum delay. System parameters can be used to model communications failure (as field equipment is more susceptible to interference than main base station communications equipment).

From an implementation viewpoint the delegation of operations undertaken by each class can be seen in Figure 6.1.

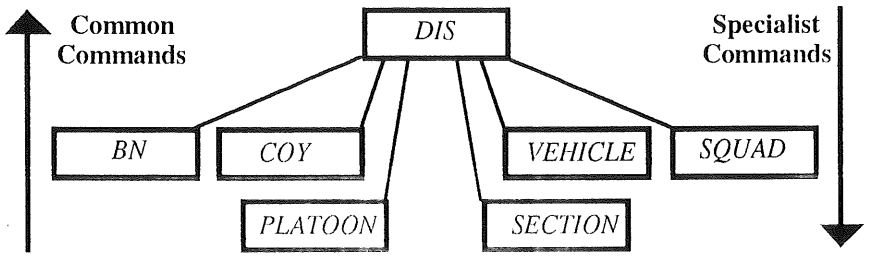

Figure 6.1

Objects of class BN (i.e: battalion) represent aggregations of individual vehicles and squads of men. Figure 6.1 shows how unique commands seek out their intended host and are executed according to the pointer type that the message request came from. Computer Scientists are concerned with correct typing of variables and classes because methods requested from a certain class must be called via a pointer of the correct type, otherwise a type mismatch error occurs.

For example, `ptrPLATOON->getUnitCommander()` will return the name of the commander for a particular platoon. The pointer variable `ptrPLATOON` only points to one specific instance of a PLATOON class and therefore operations belonging to that class are quite distinct from the same named operations belonging to another instance of a PLATOON class. The requested command may be issued at battalion level but will only be executed courtesy of a specific instance of PLATOON. If a command were issued to rename a unit (e.g: `ptrPLATOON->namePlatoon("new_name")`) then the PLATOON class would determine whether it could perform the request or pass in on to its parent class. Delegation causes the method request to move up the inheritance hierarchy where it is serviced by a generic method in the DIS class.

The two key elements responsible for organising the data structure aspect of the EMBLEM wargame are the object manager and the communications manager. These two managers have access to each object in the system and coordinate those object's actions. Lost communications are detected via the communications manager and messages are sent using the access to each object that the object manager provides. Addresses of new replacement nodes or supplemental nodes are found using the routines supplied with the object manager and each specific object. Routes through the data structure can be found using routines in the communications manager.

7 Fixed Indicators based on Error

To help with the maintenance of the communications structure we evaluate feedback from units at each level of command. The feedback can come in many forms and may only be required prior to pursuing some particular request. For example it may be necessary to determine the operational status of a unit prior to requesting a service from it (as illustrated in the source code extract in Figure 5.1). A return value of *K_Kill* indicates that the service provider (i.e: the unit) is not capable of responding. Note that other valid states for a unit in our software model are *M_Kill* (unable to move), *F_Kill* (unable to fire), *C_Kill* (unable to communicate), *C_NP* (communications temporarily not possible) and, of course, *Alive*.

Objects which are in no fit state to carry out commands can be identified simply by viewing their current state. Should a failure be noted then the player can be asked to choose an alternative node to take control of communications. Alternatively, an automated routine could spawn a new object to take over from the failed node or even select another node to act as a communications bridge.

Another error trapping strategy requires the polling of each node throughout the network at fixed intervals. Response failures could then be logged and (if necessary) a radical review of the structure could be undertaken, rebuilding and establishing the links where necessary.

Whereas one strategy for error handling is dynamic (dictated by current events) an alternate view is closely related to batch processing which entails radical reform of the network at prescribed times rather than during execution. Batch processing would temporarily bring the network 'down' whilst it were taking place. Consider Figures 7.1 and 7.2, which illustrate the difference between dynamic and batch error handling.

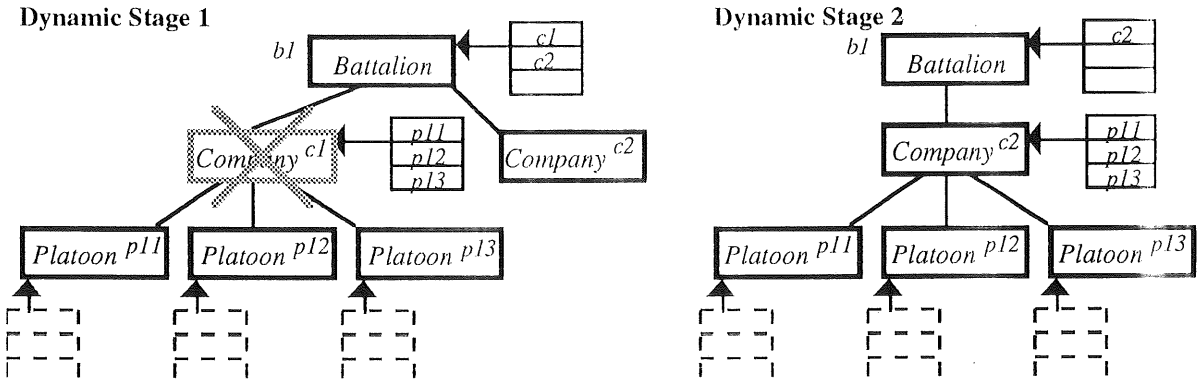

Figure 7.1

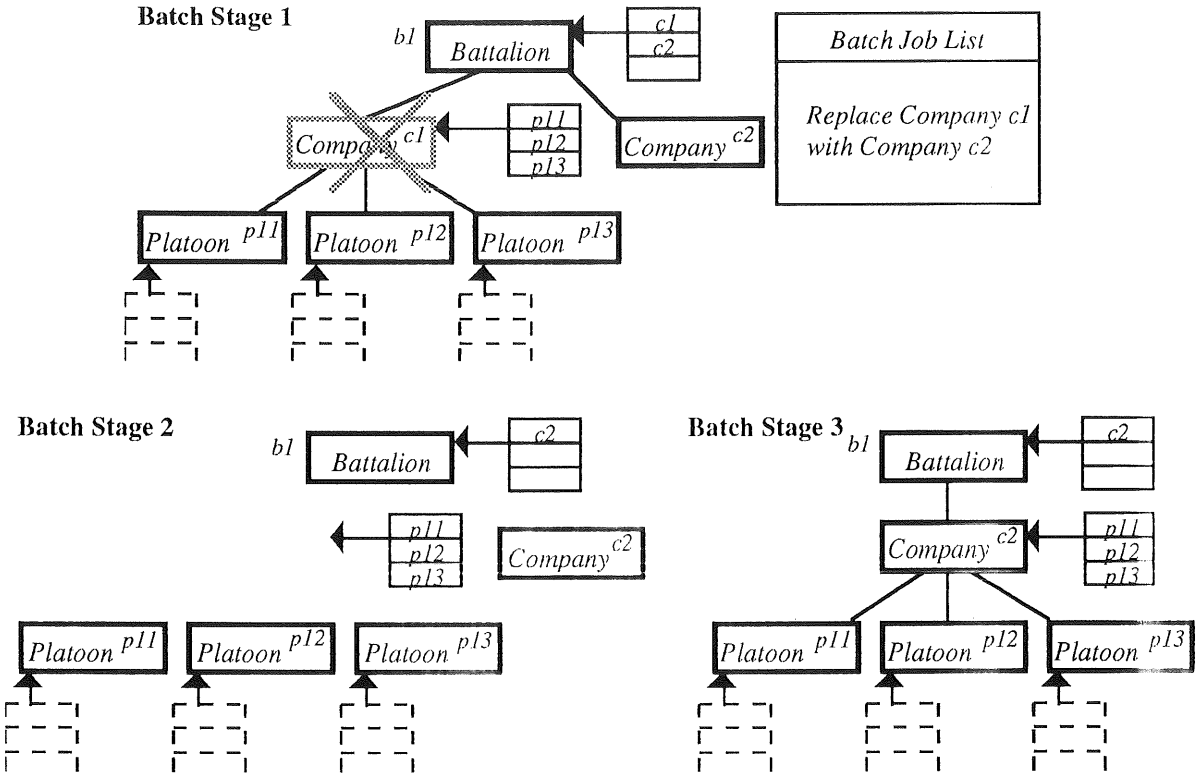

Figure 7.2

Note how the complete structure is rebuilt during stages 2 and 3 of the batch processing example. The link tables are updated and used to determine new links, thus avoiding the network's *dead zones*.

Routines to handle the storage, resurrection and allocation of nodes with new link address are required to ensure that the network structure can take care of itself, beyond the interaction required by a player of our wargame when choosing an alternate communicating unit. To ensure smooth game play within EMBLEM we adopt dynamic restructuring as it is more suited to the player's needs for continuity.

8 Degrees of Stability

The ability of our communications network to survive, despite certain node failures, has been discussed. We have also, briefly, mentioned the notion of a critical failure threshold (dictated by user-defined parameters), beyond which the entire network is in jeopardy. In our computer model, as in reality, the failure or destruction of nodes in the network reduces the effectiveness of command, either through the failure to deliver messages to certain units or by increasing the delay in message delivery.

In our software model the boundary of critical failure is reached when n subordinate nodes fail, from which the system cannot recover. These disjoint failures will eventually trigger an end-of-game

signal.

As soon as all available resources are depleted then failure of the communications structure is assured following any further node failure. We do not model the ability to rebuild damaged nodes as they are permanently removed from the command structure.

The fundamental problem with attributing a number of minor node failures with the failure of the entire system is that we cannot be sure if an accurate representation of the general failure would be caused if n nodes failed over a fixed period of time. Building any computer model will require certain assumptions to be made about areas such as the number of attempts at rebuilding or deciding when a node has been damaged enough to make it fail. Our goal is to produce a model that is a close approximation of reality. Experience with using the model and observing military command and communications situations can guide us in the assignment of parameters that govern the success and general failure of the communication system that we seek to build.

9 π-calculus Formalisation of Dynamic Military Communications

The dynamic rerouting of communications in a military structure and the subsequent software model has been the subject of the majority of this paper. We now present a formal representation of the communications network, showing how messages are routed around the network dynamically during its execution.

From a military perspective the ability to change links within an organisation during battle is essential. Dynamic rebuilding is highly desirable for military applications since combat forces cannot take a *timeout* whilst restructuring occurs.

We use the π-calculus (pronounced pi-calculus) [8] to help us in the task of specifying formally the dynamic nature of our model's data structure. The π-calculus is a process algebra which captures process behaviour; a process being a military unit in this case. Unlike a functional notation, such as Z [10], used for defining data storage and operations, process algebras concentrate on the ordered sequence of events that make up the behaviour of a process. There is no concept of timing except for the synchronisations that take place between communicating processes. Process synchronisation is a formalisation of communication along a similar channel at the same instance in time. The π-calculus itself has the ability to receive channels passed to it as parameters and then use those channels during subsequent communications to other processes in the system. Consequently, the π-calculus meets our needs in modelling dynamic connections between units. The appendix contains the full formal specification of the EMBLEM data structure together with explanatory text.

Extensions to the nature of dynamically changing process algebras have been proposed [14] and it is conceivable that further work could be undertaken to map our work with the π-calculus to other similar high order process algebras, such as [14].

One avenue to further research in the area of dynamic communicating systems is capturing the idea of adding new processes to an existing network whilst allowing existing processes in that network

to communicate with those new processes. In our present π-calculus specification only a finite number of processes are present, where the links between those processes represent the dynamically changing elements of the system. [13] reports on other notations to cope with adding new processes and changing communications links although we recognise that no suitable notation exists yet which caters for both new processes and new links during execution. We intend this section to address only part of these two issues whilst explaining the key elements of the π-calculus specification and also how this particular formal notation can be interpreted by a programming language, such as C++ [11] or various concurrent actor languages [16], both of which include the concepts of communicating objects.

Figure 9.1 shows how the processes within our structure interconnect. The labelled lines represent channels through which processes synchronise and communicate. For clarity the channel names have been abbreviated thus: *give* $\rightarrow g$, *talk* $\rightarrow t$ and *switch* $\rightarrow s$. Also, further channels from processes subscripted with n have been omitted to aid clarity.

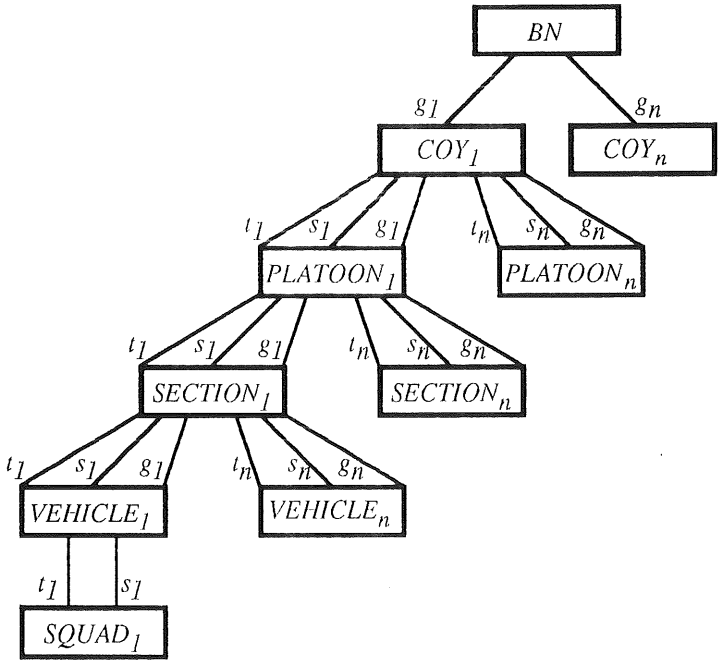

Figure 9.1

Each channel t_1 to t_n represents a talk channel and is bidirectional, representing communications propagating up and down the chain of command. Note that VEHICLE processes do not connect to SQUAD processes using the g_1 to g_n channels. This is due to the terminal node status of a SQUAD process; it has no subordinates to pass new channels down to.

The nature of each process and channel is further explained in Figure 9.2.

Action	*Description*	*Applicable Processes*
$\mu X = talk\ .\ X$	*talk repeatedly to commander or subordinates.*	*COY, PLATOON, SECTION, VEHICLE, SQUAD*
$give(t's')\ .\ \overline{switch}\ t's'\ .\ \emptyset$	*receive two new channels on give channel. Pass two new channels to subordinate process to enable comms. between subordinate and new alternate process at same level as current process then stop executing current process.*	*COY, PLATOON, SECTION VEHICLE*
$\mu X = switch(t's')\ .\ X(t',s')$	*receive new channels, then switch over to use those new channels.*	*PLATOON, SECTION, VEHICLE, SQUAD*
$\mu X = give\ t\ s\ .\ X$	*send two channels to subordinate for use by sub-subordinate.*	*COY, PLATOON, SECTION*

Figure 9.2

Due to the generic nature of the actions specified in the left-hand column of the table in Figure 9.2 recursion is defined using Hoare's definition [4, §1.2.2, p.122] in rows 1, 3 and 4.

Each process in our π-calculus specification can now be defined in terms of input and output channels, together with appropriate actions. For example, the formal definition of the BN process (representing the battalion node) is given in π-calculus notation as:

$$BN \stackrel{def}{=} \overline{give_i}\ talk_j\ switch_j\ .\ BN$$
$$(i \leq gMaxCOY \wedge j \leq gMaxPLATOON \wedge (state(BN) \neq K\_Kill \wedge state(BN) \neq C\_Kill \wedge state(BN) \neq C\_NP))$$

Figure 9.3

An invariant may be included with a process definition which ranges over the process during its lifetime. In Figure 9.3 the invariant states that the subscript for any BN channel *give* must be within the range 1 to `gMaxCOY` (a global variable value). Also, *talk* and *switch* parameters must not exceed the total number of available PLATOON processes in the system. Lastly, the invariant checks the state of the process to ensure that it is capable of receiving communications.

Basically, the behaviour of process BN is to 'give' the parameters $talk_j$ and $switch_j$ to a subordinate COY process, which in turn passes them on to a PLATOON process to start using. When a process switches between one channel and another (possibly during rerouting) the new channel will appear in the input parameter $switch(t',s')$. After recursion the process will start to use the new channel that was passed to it as a parameter during the previous communication.

The appendix contains the complete π-calculus specification of the EMBLEM data structure and explains the details pertaining to processes at each level of command. Discussion about the contents and behaviour of the processes can also be found in the appendix.

The semantics of the π-calculus provides us with the ability to switch communication channels and is ideally suited to our needs. We use the mathematical rigour of this formal specification language

to enable us to reason about the detailed design of the data structure and provide an abstract representation of the communications model.

10 Conclusions

The ability to adapt to a changing environment is a necessity for any organisation fighting to survive. Changes that occur within the organisation (functional changes) and those which are forced upon the organisation from the external environment need to be dealt with by a system of change that is built into the structure of the organisation itself. We introduce such a data structure that forms the heart of our computer model wargame; EMBLEM.

This paper has illustrated a dynamic communications network, represented as a data structure and based upon a military organisation. Clearly, comparisons can be drawn between the military command structure and that of civilian emergency services, such as Fire, Ambulance and Police forces. To a lesser extent we can also map our model of military command onto an industrial and commercial management structure.

The need for adaptive structures is paramount within organisations that deal with life threatening situations. The introduction of a communications network that is adaptive allows our computer model to cope with a degree of failure throughout each of the six layers of command that we have identified. Nodes on the network have the ability to reroute their messages, enabling the network to continue to function. In our software model existing nodes act as bridges between one layer and another, sharing the load of some failed node of the same class. Links are kept active provided that the maximum number of subordinate links-per-node are not exceeded.

An extension to our original strategy of rerouting communications through existing nodes is presented. We introduce backup nodes that can be allocated (up to some user defined maximum). These extra nodes (of classes COY, PLATOON and SECTION) take control of failed node communications. Upon exhaustion of these backup nodes our original strategy of sharing communications through existing nodes comes into effect. Consequently, a two tier redundant communications system is proposed; namely that of rerouting communications and allocating backup nodes to substitute failed nodes.

Ultimately we seek to provide a flexible model of communication that can change throughout its lifetime and mimic closely the command and control communications in a military hierarchy. The military structures that we model are subjected to hostile environments.We therefore need to model the effects of such environments. We give our software model a certain amount of redundancy but insist that failed nodes cannot be repaired.

We have shown an efficient, yet simple model for a communicating organisation that can deal with a certain amount of internal and external change. Our studies reveal that redundancy must be incorporated into the very structure of the organisation itself if it is to be flexible enough to withstand certain changes. Further research is needed to map our simple model onto larger organisations. Also

we seek to expand upon the ideas of prioritising messages and reusing repaired nodes after their initial failure.

Although we do not explicitly discuss other non-military organisations the basic structure of our data model shares many attributes found in industry and commerce. We can translate the layers of military command into equivalent layers of management. Consequently, our work has application in commercial and industrial sectors and can be used to show that computerised communication models, such as that used in EMBLEM, can be of benefit to those wishing to model change in any hierarchical structure with a similar form.

References:

[1] Bailey, M., Kemple, W. and West, M. et al. (1994). Object-Oriented Modelling of Military Communications Networks. *Journal of the Operational Research Society*. v45, no.10. pp1108 - 1122.

[2] Gallagher, M.A. and Kelly, E.J. (December 1991). A New Methodology for Military Force Structure Analysis. *Operations Research*. v39, no.6. pp877 - 885.

[3] Hartley III, D.S. (August 1992). Military Operations Research: Presentations at ORSA/TIMS Meetings. *Operations Research*. v40, no.4. pp640 - 646.

[4] Hoare, C.A.R, (1985). *Communicating Sequential Processes*. Prentice-Hall: London.

[5] IEEE Standards Board. (March 1993). *IEEE Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications*. IEEE Std. 1278 - 1993.

[6] Meyer, B. (1988). *Object oriented software construction*. Prentice-Hall International.

[7] Milner, R., (1989), *Communication and Concurrency*. Prentice-Hall: London.

[8] Milner, R. (October 1991). *The Polyadic π-Calculus: A Tutorial*. LFCS Report **ECS-LFCS-91-180**. Department of Computer Science, University of Edinburgh.

[9] Singh, G. and Bommareddy, M. (June 1994). Replica Placement in a Dynamic Network. *Proceedings of International Conference on Distributed Computing Systems*. IEEE, Poznan, Poland, June 21 - 24, 1994. Conf. Code 21451. pp528 - 535.

[10] Spivey, M. (1989). *The Z Notation*. Prentice-Hall: Hemel-Hempstead.

[11] Stroustrup, B. (1991). *The C++ Programming Language*, (2nd. ed.). Addison-Wesley, Reading: MA.

[12] Tanenbaum, A.S. (1988). *Computer Networks,* (2nd ed.). Prentice-Hall: Englewood Cliffs.

[13] Taylor, P.N. (May 1995). *The Analysis of Formal Models of Communication for the Specification of Reusable Systems*. University of Hertfordshire, Computer Science Division, Ph.D. Transfer Report.

[14] Thomsen, B., (1990). *Calculi for higher-order communicating systems*. Ph.D. Thesis. Imperial College, London University.

[15] Wegner, P. (1987). *The Object-Oriented Classification Paradigm*. Research Directions in Object-Oriented Programming. B. Shriver and P. Wegner (Editors). MIT Press.

[16] Yonezawa, A and Tokoro, M. (Eds.) (1987). *Object-Oriented Concurrent Systems*. MIT Press.

Appendix:

This appendix presents the complete π-calculus specification of the EMBLEM communicating data structures. A formal representation of the network is given, showing how messages are routed around the network dynamically during its execution.

A brief summary of π-calculus terms are given, using the definition of the BN process in Figure A.1 as an example.

$$BN \stackrel{def}{=} \overline{give_i}\ talk_j\ switch_j\ .\ BN$$
$$(i \le gMaxCOY \ ^{a}\ j \le gMaxPLATOON \ ^{a}\ (state(BN) \ne K\_Kill \ ^{a}\ state(BN) \ne C\_Kill \ ^{a}\ state(BN) \ne C\_NP))$$

Figure A.1

Firstly, processes definitions are read left to right. Process actions which have a line above them denote output channels (e.g: $\overline{action}$). Any actions immediately following an output channel are parameters for that output channel. In Figure A.1 both $talk_j$ and $switch_j$ actions are parameters of the output on channel $\overline{give_i}$. A full stop (.) separates actions, denoting the ordered sequence of actions that dictate a process's behaviour. The symbol + denotes choice in the π-calculus (see Figure A.2), where some decision is made regarding the use of one of a possible number of action sequences offered by a process. Two types of choice exist, deterministic and nondeterministic. These relate to whether the environment can influence which action sequence is chosen. We do not expand on this particular topic

here. The reader is referred to [4, §1.1.3, p29 & §3, p101] and [7, §1.2 p20 & §2.3, p42] for more information regarding determinism. Recursion is common in process definitions and is shown as a reference to the process's name at the end of each expression. Without recursion a process would simply execute once and then terminate.

Input channels, such as those in Figure A.2, use braces to denote their parameters (e.g: $give_k(t',s')$).

Expansion of the specification leads to the definition of the COY process itself. This class of process extends the behaviour of BN processes by: (i) adding the ability to talk to neighbouring processes at level *n-1* and level *n+1*, and (ii) receiving and forwarding new channels for the subordinate PLATOON processes to use directly, rather than forwarding messages for *n+2* processes to use (such as occurs in process BN).

Ex.1 $COY(t,s,g) \stackrel{def}{=} t\ .\ COY(t,s,g)$ (1)

$+\ g(t',s')\ .\ \overline{st}'s'\ .\ \emptyset$ (2)

$+\ \overline{g_i}\ t_j\ s_j\ .\ COY(t,s,g)$ (3)

$(i \leq gMaxPLATOON \wedge j \leq gMaxSECTION \wedge$

$(state(COY) \neq K\_Kill \wedge state(COY) \neq C\_Kill \wedge state(COY) \neq C\_NP))$

Ex.2 $COY_k(t,s,g) \stackrel{def}{=} COY(talk_k, switch_k, give_k)$ (1)

$+\ \overline{g_i}\ t_j\ s_j\ .\ COY(t,s,g)$ (2)

$(k \leq gMaxCOY \wedge i \leq gMaxPLATOON \wedge j \leq gMaxSECTION \wedge$

$(state(COY_k) \neq K\_Kill \wedge state(COY_k) \neq C\_Kill \wedge state(COY_k) \neq C\_NP))$

Ex.3 $COY_k \equiv talk_k\ .\ COY_k + give_k(t',s')\ .\ \overline{switch_k}\ t's'\ .\ \emptyset$ (1)

$(k \leq gMaxCOY \wedge$

$(state(COY_k) \neq K\_Kill \wedge state(COY_k) \neq C\_Kill \wedge state(COY_k) \neq C\_NP))$

Figure A.2

In Figure A.2, Ex.1(1) defines the signature of the process COY together with the initial action choice of talking to either of its neighbours in the command structure. Ex.1(2) specifies the receipt and forwarding of new *talk* and *switch* channels, after which COY terminates. These channels are sent to a process of class PLATOON. Ex.1.(3) duplicates the behaviour of BN by sending *talk* and *switch* channels to a PLATOON process for subsequent use by a SECTION process (level *n+2* from COY). The invariant $(i \leq MaxPLATOON \wedge j \leq MaxSECTION)$ limits the range of subscripts to the total number of PLATOON and SECTION processes respectively. Again, as with other processes in our formal model the state of the process is queried to ensure that it can receive communications.

The extension of COY in Figure A.2 with Ex.2 and Ex.3 simply assigns the true meanings of

the actions and channels, enabling the specification of numerous COY processes in the completed specification. The subscript k is used to limit the total number of COY processes to some global value.

Figure A.3 expands the invariant seen in other processes to check the fighting status of the squad attached to a vehicle. If either the crew or the squad for the vehicle are in a fit state then communications can proceed.

Ex.1 $VEHICLE(t,s,g) \stackrel{def}{=} t\ .\ VEHICLE(t,s,g)$ (1)

$+\ g(t',s')\ .\ \overline{st's'}\ .\ \emptyset$ (2)

$+\ s(t',s')\ .\ VEHICLE(t',s',g)$ (3)

Ex.2 $VEHICLE_k(t,s,g) \stackrel{def}{=} VEHICLE(talk_k,switch_k,give_k)$ (1)

$+\ s(t',s')\ .\ VEHICLE(t',s',g)$ (2)

$(k \leq gMaxVEHICLE \wedge (state(VEHICLECREW_k) \neq K\_Kill \wedge state(VEHICLECREW_k) \neq C\_Kill \wedge state(VEHICLECREW_k) \neq C\_NP) \wedge (state(vehicleSquad(VEHICLE_k)) \neq K\_Kill \wedge state(vehicleSquad(VEHICLE_k)) \neq C\_Kill \wedge state(vehicleSquad(VEHICLE_k)) \neq C\_NP))$

Ex.3 $VEHICLE_k \equiv talk_k\ .\ VEHICLE_k + give_k(t',s')\ .\ \overline{switch_k}\ t's'\ .\ \emptyset$ (1)

$(k \leq gMaxVEHICLE \wedge (state(VEHICLECREW_k) \neq K\_Kill \wedge state(VEHICLECREW_k) \neq C\_Kill \wedge state(VEHICLECREW_k) \neq C\_NP) \wedge (state(vehicleSquad(VEHICLE_k)) \neq K\_Kill \wedge state(vehicleSquad(VEHICLE_k)) \neq C\_Kill \wedge state(vehicleSquad(VEHICLE_k)) \neq C\_NP))$

Figure A.3

The behaviour of VEHICLE is similar to that of COY in Figure A.1. Differences between the two processes come in the form of: (i) the removal of output on *give* channel, as a SQUAD process has no subordinate and (ii) the extension of the invariant (as described above). SQUAD is similar again, except for the removal of the receiving channel $give(t',s')$.The extension of the invariant to monitor the state of the squad still applies. Note that only an interrogation of the actual command can detect whether an *M_Kill* vehicle or squad is requested to move and an *F_Kill* vehicle or squad is requested to engage the enemy. Formally, we only model the fact that the processes must not be in a *K_Kill, C_Kill* or *C_NP* state.

Ex.1 $SQUAD(t,s) \stackrel{def}{=} t \,.\, SQUAD(t,s)$ (1)

$+\ s(t',s') \,.\, SQUAD(t',s')$ (2)

Ex.2 $SQUAD_k(t,s,g) \stackrel{def}{=} SQUAD(talk_k, switch_k)$ (1)

$+\ s(t',s') \,.\, SQUAD(t',s')$ (2)

$(k \leq gMax\ SQUAD \wedge (state(SQUAD_k) \neq K\_Kill \wedge state(SQUAD_k) \neq C\_Kill \wedge state(SQUAD_k) \neq C\_NP))$

Ex.3 $SQUAD_k \equiv talk_k \,.\, SQUAD_k + \overline{switch_k}(t's') \,.\, SQUAD_k$ (1)

$(k \leq gMax\ SQUAD \wedge (state(SQUAD_k) \neq K\_Kill \wedge state(SQUAD_k) \neq C\_Kill \wedge state(SQUAD_k) \neq C\_NP))$

Figure A.4

The complete π-calculus specification of the command structure in EMBLEM now follows. The reader is expected to apply the discussion in the first part of the appendix to the following process definitions. Certain comparisons can be drawn between the behaviour of many of the processes in the system.

BN Process:

Ex.1 $BN \stackrel{def}{=} \overline{give_i}\ talk_j\ switch_j \,.\, BN$ (1)

$(i \leq gMaxCOY \wedge j \leq gMaxPLATOON \wedge (state(BN) \neq K\_Kill \wedge state(BN) \neq C\_Kill \wedge state(BN) \neq C\_NP))$

COY Process:

Ex.2 $COY(t,s,g) \stackrel{def}{=} t \,.\, COY(t,s,g)$ (1)

$+\ g(t',s') \,.\, \overline{st}'s' \,.\, \emptyset$ (2)

$+\ \overline{g_i}\ t_j\ s_j \,.\, COY(t,s,g)$ (3)

$(i \leq gMaxPLATOON \wedge j \leq gMaxSECTION \wedge (state(COY) \neq K\_Kill \wedge state(COY) \neq C\_Kill \wedge state(COY) \neq C\_NP))$

Ex.3 $COY_k(t,s,g) \stackrel{def}{=} COY(talk_k, switch_k, give_k)$ (1)

$+\ \overline{g_i}\ t_j\ s_j \,.\, COY(t,s,g)$ (2)

$(k \leq gMaxCOY \wedge i \leq gMaxPLATOON \wedge j \leq gMaxSECTION) \wedge (state(COY_k) \neq K\_Kill \wedge state(COY_k) \neq C\_Kill \wedge state(COY_k) \neq C\_NP))$

Ex.4 $COY_k \equiv talk_k \,.\, COY_k + give_k(t',s') \,.\, \overline{switch_k}\ t's' \,.\, \emptyset$ (1)

$(k \leq gMaxCOY) \wedge$

$(state(COY_k) \neq K\_Kill \wedge state(COY_k) \neq C\_Kill \wedge state(COY_k) \neq C\_NP))$

PLATOON Process:

Ex.5 $PLATOON(t,s,g) \stackrel{def}{=} t\ .\ PLATOON(t,s,g)$ (1)

$+\ g(t',s')\ .\ st's'\ .\ \emptyset$ (2)

$+\ s(t',s')\ .\ PLATOON(t',s',g)$ (3)

$+\ \overline{g_i}\ t_j\ s_j\ .\ PLATOON(t,s,g)$ (4)

$(i \leq gMaxSECTION \wedge j \leq gMaxVEHICLE) \wedge$

$(state(PLATOON) \neq K\_Kill \wedge state(PLATOON) \neq C\_Kill \wedge state(PLATOON) \neq C\_NP))$

Ex.6 $PLATOON_k(t,s,g) \stackrel{def}{=} PLATOON(talk_k,switch_k,give_k)$ (1)

$+\ s(t',s')\ .\ PLATOON(t',s',g)$ (2)

$+\ \overline{g_i}\ t_j\ s_j\ .\ PLATOON(t,s,g)$ (3)

$(k \leq gMaxPLATOON \wedge i \leq gMaxSECTION \wedge j \leq gMaxVEHICLE) \wedge$

$(state(PLATOON_k) \neq K\_Kill \wedge state(PLATOON_k) \neq C\_Kill \wedge state(PLATOON_k) \neq C\_NP))$

Ex.7 $PLATOON_k \equiv talk_k\ .\ PLATOON_k + give_k(t',s')\ .\ \overline{switch_k}\ t's'\ .\ \emptyset$ (1)

$(k \leq gMaxPLATOON) \wedge$

$(state(PLATOON_k) \neq K\_Kill \wedge state(PLATOON_k) \neq C\_Kill \wedge state(PLATOON_k) \neq C\_NP))$

SECTION Process:

Ex.8 $SECTION(t,s,g) \stackrel{def}{=} t\ .\ SECTION(t,s,g)$ (1)

$+\ g(t',s')\ .\ \overline{st}'s'\ .\ \emptyset$ (2)

$+\ s(t',s')\ .\ SECTION(t',s',g)$ (3)

$+\ \overline{g_i}\ t_j\ s_j\ .\ SECTION(t,s,g)$ (4)

$(i \leq gMaxVEHICLE \wedge j \leq gMaxSQUAD) \wedge$

$(state(SECTION) \neq K\_Kill \wedge state(SECTION) \neq C\_Kill \wedge state(SECTION) \neq C\_NP))$

Ex.9 $SECTION_k(t,s,g) \stackrel{def}{=} SECTION(talk_k,switch_k,give_k)$ (1)

$+\ s(t',s')\ .\ SECTION(t',s',g)$ (2)

$+\ \overline{g_i}\ t_j\ s_j\ .\ SECTION(t,s,g)$ (3)

$(k \leq gMaxSECTION \wedge i \leq gMaxVEHICLE \wedge j \leq gMaxSQUAD) \wedge$

$(state(SECTION_k) \neq K\_Kill \wedge state(SECTION_k) \neq C\_Kill \wedge state(SECTION_k) \neq C\_NP))$

Ex.10 $SECTION_k \equiv talk_k\ .\ SECTION_k + give_k(t',s')\ .\ \overline{switch_k}\ t's'\ .\ \emptyset$ (1)

$(k \leq gMaxSECTION)$

$(state(SECTION_k) \neq K\_Kill \wedge state(SECTION_k) \neq C\_Kill \wedge state(SECTION_k) \neq C\_NP))$

VEHICLE Process:

Ex.11 $VEHICLE(t,s,g) \stackrel{def}{=} t\ .\ VEHICLE(t,s,g)$ (1)

$+\ g(t',s')\ .\ \overline{st}'s'\ .\ \emptyset$ (2)

$+\ s(t',s')\ .\ VEHICLE(t',s',g)$ (3)

Ex.12 $VEHICLE_k(t,s,g) \stackrel{def}{=} VEHICLE(talk_k,switch_k,give_k)$ (1)

$+\ s(t',s')\ .\ VEHICLE(t',s',g)$ (2)

$k \leq gMaxVEHICLE \wedge (state(VEHICLECREW_k) \neq K\_Kill \wedge state(VEHICLECREW_k) \neq C\_Kill \wedge state(VEHICLECREW_k) \neq C\_NP) \wedge (state(vehicleSquad(VEHICLE_k)) \neq K\_Kill \wedge state(vehicleSquad(VEHICLE_k)) \neq C\_Kill \wedge state(vehicleSquad(VEHICLE_k)) \neq C\_NP)$

Ex.13 $VEHICLE_k \equiv talk_k\ .\ VEHICLE_k + give_k(t',s')\ .\ \overline{switch_k}\ t's'\ .\ \emptyset$ (1)

$k \leq gMaxVEHICLE \wedge (state(VEHICLECREW_k) \neq K\_Kill \wedge state(VEHICLECREW_k) \neq C\_Kill \wedge state(VEHICLECREW_k) \neq C\_NP) \wedge (state(vehicleSquad(VEHICLE_k)) \neq K\_Kill \wedge state(vehicleSquad(VEHICLE_k)) \neq C\_Kill \wedge state(vehicleSquad(VEHICLE_k)) \neq C\_NP)$

SQUAD Process:

Ex.14 $SQUAD(t,s) \stackrel{def}{=} t\ .\ SQUAD(t,s)$ (1)

$+\ s(t',s')\ .\ SQUAD(t',s')$ (2)

Ex.15 $SQUAD_k(t,s,g) \stackrel{def}{=} SQUAD(talk_k,switch_k)$ (1)

$+\ s(t',s')\ .\ SQUAD(t',s')$ (2)

$k \leq gMax\ SQUAD \wedge state(SQUAD_k) \neq K\_Kill \wedge state(SQUAD_k) \neq C\_Kill \wedge state(SQUAD_k) \neq C\_NP$

Ex.16 $SQUAD_k \equiv talk_k\ .\ SQUAD_k + \overline{switch_k}(t's')\ .\ SQUAD_k$ (1)

$k \leq gMax\ SQUAD \wedge state(SQUAD_k) \neq K\_Kill \wedge state(SQUAD_k) \neq C\_Kill \wedge state(SQUAD_k) \neq C\_NP$