Modeling Cooperative, Reactive Behaviors on the Battlefield with Intelligent Agents

Dr. Thomas R. Ioerger
Dr. Richard A. Volz
Dr. John Yen
Department of Computer Science
Texas A&M University
301 H.R. Bright Building
College Station, TX 77843-3112
(979)845-5534
{ioerger,volz,yen}@cs.tamu.edu

ABSTRACT: *More believable and intelligent control of entities and aggregates is needed in military simulations for prediction, evaluation, planning, and training in the area of battlefield operations. Intelligent agents are a technology that can be used to achieve this level of believable behavior in a simulation autonomously by modeling various aspects of the complex goal-oriented decision making of the units involved. While some simpler behaviors at the entity level can currently be simulated quite well (e.g. route-planning for a single tank platoon), the behavior of larger aggregates, such as battalions and brigades, are considerably more complex, requiring information-gathering, situation assessment, resource allocation, coordination with other units, etc. In the University XXI project, we have been developing an intelligent agent architecture to support the training of tactical operations center (TOC) staff officers at the brigade level. We have focused on simulation of the behavior of battalions, to which the brigade commander issues orders and with which many of the personnel in the TOC interact (e.g. to provide information or support). Our simulation of battalions inter-operates with OneSAF Testbed (OTB) to control the friendly units at the battalion level, and to receive responses from the enemy in real-time. In this paper, we describe the underlying agent architecture we have developed for simulating battalion behavior. There are three central components: a generic task representation language for capturing procedural knowledge about when and how to carry out various functions within the battalion TOC, an agent algorithm (interpreter) for carrying out the execution of these tasks in an interleaved way that allows for sufficiently reactive behavior, and an implementation of this architecture in Java and Jess (Java Expert System Shell). We have used our task representation language to do a large amount of knowledge acquisition about staff functions in a battalion TOC by encoding doctrine (e.g. techniques, tactics, and procedures) acquired through reading documentation/manuals and through interviews with military expert. We have implemented a prototype of our system on a Movement-To-Contact scenario. The success of our approach validates the potential utility of intelligent agent technology for these types of military applications, and our system will be further developed as the basis of simulation-based training system for digital forces staff operations. The main contribution of this research has been to provide insight on how agents can be used to model specific functions within a sophisticated aggregate such as a battalion, and to effectively make decisions that synthesize both scenario-specific orders as well as general knowledge about how to adaptively carry out those functions as a situation diverges from expectations.*

1. Introduction

Simulations of battlefield behavior play a variety of important roles in the modern military [1]. Such simulations can be used to evaluate courses of action, by illustrating how opposing forces (OPFOR) are likely to respond, to evaluate the impact of new equipment or strategies, or to plan for contingencies (i.e. "what-if" analyses). In addition, battlefield simulations are playing an increasing role in training, especially for staff officers who need to learn how to make decisions under various circumstances and to work with new digital equipment (e.g. ABCS).

Though a variety of simulations exist, they are often limited in the realism of the behavior they can model. Early simulations allowed for only "canned" or "scripted" scenarios, in which a specific situation was re-played, and trainees were asked to make decisions, though they did not directly affect the outcome of the run. To enable a dynamic response by the OPFOR, other simulations used a networked approach where human assistants manipulated OPFOR units from separate consoles. More recently, simulations such as MODSAF have allowed behaviors for simple units on the battlefield to be automated via programmed routines, such as finding routes for tanks given current position and a target location to reach while continuously firing on nearby enemy positions. These simulations have tended to only capture simple behaviors at the entity level. However, there is also a great need to automate behaviors at the aggregate level, such as company, battalion, and higher.

This is needed for both friendly forces - for example to train a brigade staff without requiring a large number of individuals to simulate the carrying out of commands - and opposing forces, to make intelligent opponents that work together and are challenging to defeat.

The difficulty in automating the behaviors of these larger units is that their behaviors on the battlefield are much more complex, involving a variety of forms of decision-making. On the one-hand, they have to follow orders given to them by their commanding units. On the other hand, op-orders are not always specific or perfect, and the enemy might do something unexpected. For these reasons, aggregate entities also need to be flexible and react appropriately to situations as they develop.

Simulating this decision-making presents an important opportunity for intelligent agents to play a role in battlefield simulations. Intelligent agents are software entities that are interact with an environment (i.e. can sense the state of the world, and take actions to change it), and furthermore are goal-oriented and autonomous [2]. Agents are often based on artificial intelligence (AI) techniques to make intelligent decisions and select actions that will achieve their goals. Useful techniques include: heuristic search, planning, knowledge representation, automated deduction, and management of uncertainty. For example, an agent may have a knowledge base containing rules about the battlefield that it can use to make inferences, as well as knowledge about its operators that it can use to reason about the consequences of its actions. However, it is important to emphasize that a simple approach such as immediately taking actions whenever certain conditions are met (e.g. trigger rules) is inadequate; the agents must be capable of forming and carrying out multi-step plans that extend over time, which are necessary to accomplish more complex goals.

For the types of applications we are interested in, we can assume that the battle is taking place under fairly well-defined conditions where we know the terrain, the general location and size of OPFOR, probably avenues of approach, etc. (i.e. METT-TC). Hence we can assume the simulation will have the benefit of already having op-orders that lay out the plan of action for each unit, and we can avoid the more complex issues of constructing the plan and dealing with arbitrary tactical situations from scratch (i.e. not full MDMP). However, we also want the agent-controlled units to avoid blindly following the prescribed course of action without regard to the situation around them. For believability, it is essential that units adapt by: taking advantage of targets of opportunity, finding alternative methods of maneuver around obstacles like destroyed bridges, responding to ambushes, calling for backup when out-numbered, etc. Thus the challenge is how to control aggregates so that they carry out specified orders under nominal conditions, but autonomously develop and take alternative courses of actions when and where appropriate.

In addition to making decisions on their own, it is crucial in some simulations to have units interact. For example, on a real battlefield, the S2 (intelligence) and S3 (maneuver) officers in a battalion TOC, will be in constant contact with their counterparts at the brigade level (higher), the company level (lower), and adjacent battalions (laterally), as well as talking with each other and other officers in the same TOC. Such interactions are essential to realistic simulation of friendly forces; no aggregate unit operates on its own, but instead they are all highly coordinated and interacting. This is exactly the aspect of behavior which is useful for a digital battlestaff trainer. However, no current systems are capable of simulating this kind of communication/interaction and its effect on the behaviors of the entities involved.

In this paper, we describe our initial work on the University XXI project, in cooperation with Ft. Hood and sponsored through the Department of Defense. In the preliminary phase of this project, we are tasked with developing an intelligent-agent architecture that can make intelligent decisions for units on the battlefield - in particular, battalion TOCs - such that the battalions carry out orders, cooperate with each other and brigade, and sufficiently react to situations that deviate from expectations. The system is built on top of OneSAF Testbed (OTB), of which we can monitor the state and issue commands to control entities. On the other end, the system provides an interface for brigade staff officers to interact, including communicating with simulated battalions, requesting status reports, issuing commands (e.g. FRAGOs), responding to battalion requests for information, support, or resources, etc. (This is only part of a digital battlestaff trainer.)

The system itself consists of an intelligent agent (one agent for each battalion) that makes decisions about what to do and how to communicate in the context of a battle. The agent uses a large knowledge base of procedural knowledge about how to carry out various functions within a TOC, gathered from manuals (TTP, doctrine) and from interviewing military experts. This knowledge is encoded in a special knowledge representation language we have devised, called TRL (Task Representation Language). The agent uses a sophisticated algorithm to infer what actions to do at each time step, which involves: tracking its progress through an initial plan, selecting the most preferred method for a task given the circumstance, maintaining multiple tasks in parallel (such as attacking in one direction while monitoring enemy approaching from another direction, all the while responding to requests and

commands, and submitting reports), properly terminating tasks when conditions change and starting more appropriate tasks in their place, and so on.

In the remainder of this paper, we describe three core pieces of technology we have developed for this project: the task representation language, the agent algorithm, and our implementation of the system in Java, Jess, and RMI (Remote Method Invocation). Then we discuss our experience with applying this system to simulating a Movement-to-Contact scenario. Finally, we discuss some strengths and limitations of the approach, and directions for future work. This agent-based approach can be used to effectively model cooperative, reactive behavior in other aggregate units in military simulations for generating believable actions and interactions in a wide variety of situations where doctrine provides reasonably well-defined procedures for carrying out tasks and responding to events.

2. Task Representation Language

A central component of the system we are building is the knowledge representation language we use for encoding procedural information about how to carry out various battle-staff functions. For this purpose, we have developed a specialized language called TRL, for Task Representation Language. TRL allows the description of four fundamental types of knowledge: goals, tasks, methods, and operators. Here are the intuitive meanings of these terms in our framework:

- **Goals** are conditions to be achieved, such as *enemy-defeated* or *brigade-informed*.
- **Tasks** are higher-level actions to be done, such as *surround-enemy* or *track-enemy*.
- **Methods** are descriptions of specific procedures that can be used to accomplish tasks, which can refer to sub-tasks or operators.
- **Operators** are primitive actions (from the perspective of TRL), such *move* or *fire*, which can be directly executed in the environment, along with communications (RFIs, status reports, call for fire support...) with brigade and other units.

In TRL, most of these elements can have pre-conditions (which must be satisfied to initiate them) and termination conditions (which can interrupt the process if they become true while it is executing). In addition, priorities or preference conditions can be associated with methods, to help decide which one is the best to use to carry out a task in a given situation.

TRL combines the features of several other action languages used in AI planning systems. It has operators with pre-conditions like the standard STRIPS formalism [3] (though post-conditions or effects are not explicitly represented, because the agent algorithm below does not perform goal regression). However, it is more similar to hierarchical planning and task-decomposition languages [4,5], especially because of the recursive way that tasks depend on methods, which depend on sub-tasks. And finally, TRL adds syntax for flexibly representing processes similar to systems like PRS [6], including sequences, parallel constructs, conditionals, loops, etc. In the rest of this section, we provide an overview of the syntax and semantics of TRL.

The syntax of TRL is based on descriptors written in the form of S-expressions (nested lists of symbols, derived from LISP conventionality). At the head of each list is usually a keyword indicating the type of information in that list. Keywords are prefixed with a colon for distinction, such as `:TASK` and `:METHOD`. Each descriptor contains a list consisting of: a keyword specifying the type of the descriptor, the name of the goal, task, method, or operator, a list of its formal parameters (variable names for input arguments), an optional description string, and then some additional elements unique to each. Variables in TRL are prefixed with a question mark, such as `?enemy-id` or `?x-position`. The ability to have variables in definitions of tasks and methods, etc. was found to be important for flexibly encoding procedural knowledge in generalized but compact form as schemas that did not have to be repeated for each instantiation (e.g. each unit that would carry out the task). Comments are indicated by a semi-colon (';'), which causes disregard of all the text to the end of a line. We now describe each type of descriptor and their unique elements using Baukus-Naur Form (BNF) grammar.

Goals are described by giving a condition to achieve, and specifying tasks that can be used to achieve those conditions:

```
<GOAL> ::=
  (:GOAL <name> (<variables>* )
     (:COND <condition>)
     (:TASK <name> <value>* ) )
```

where <name>s are symbols, and <value>s are symbols, numbers, or variables. The first list of variables are the formal parameters of the goal, and the second list of values are input arguments supplied to the relevant task. Only one task may be associated with a goal; any additional decision-making would be done through method selection. Conditions are structured elements that are used widely in other descriptors as well. They are specified by lists that contain the keyword `:COND` followed by a list of predicates, which are themselves lists containing predicate names and argument values. In addition, predicates can be negated by enclosing them in a

list with the keyword :NOT. The list of predicates is explicitly treated as a conjunction ('and'). Empty conditions are interpreted as always true; the constants TRUE and FALSE may be used as well.

```
<condition> ::= [ TRUE | FALSE |
          <predicate> | (:NOT <predicate>) ]*

<predicate> ::= (<name> <value>+ )
```

While goals play a relatively minor role in our simulations (just at the top-level to initiate the scenario), tasks and methods constitute the majority of the knowledge we have captured to date. Task descriptions primarily specify a set of possible methods that can be used to carry them out (along with other relevant information such as termination conditions). Syntactically, a task contains the keyword :TASK, the task name, some arguments, any termination conditions[1], and then the method specification. Aside from naming and providing input arguments to methods, they may also be assigned priorities or preference conditions.

```
<TASK> ::=
  (:TASK <name> (<variables>* )
    (:TERM-COND <condition>)
    [ (:METHOD (<name> <value>*)
        [(:PRIORITY <int>) |
         (:PREF-COND <condition>) ] ) ]+ )
```

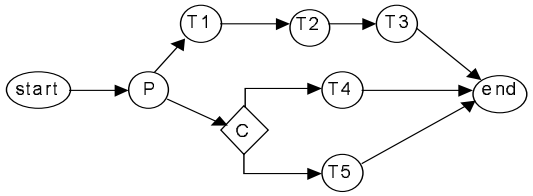
If multiple methods are defined, then one is selected based on the following resolution scheme. Essentially, the set of methods is sorted by two keys, the first being the preference condition and the second being the priority. There are three possible values for the preference condition: 1) condition satisfied, 2) no preference condition specified, and 3) preference condition not satisfied. Their sort order is that given above. The priority sort order is simply the numeric sort order of integers. Preference conditions are not mandatory; if all else fails, methods whose preference conditions are not satisfied may still be used. These systems form a two-level ranking scheme, though they can be used independently as well (e.g. only priorities or only preference conditions). Methods that still tie by all of these rules are chosen in order, top-down. Here is an example of a task description:

```
(:Task attack-enemy (?company-id ?enemy-id)
  (:Term-cond (enemy-destroyed ?enemy-id))
  (:Method (call-for-indirect-fire
             ?company-id ?enemy-id)
    (:Pref-cond (have-priority-of-fire
                  ?company-id)))
  (:Method (attack-enemy-with-mortars
             ?company-id ?enemy-id)
    (:Pref-cond (small ?enemy-id)
          (have-mortar-assets ?company)
          (< (dist (position ?company-id)
             (position ?enemy-id))7-km))
  (:Method (attack-enemy-by-direct-fire
             ?company ?enemy)
    (:Pref-cond FALSE)))
```

Note that attack-by-direct-fire is always a last resort because its preference condition is always false and it is last in the list. Of course, many more methods and preference conditions could be added to this example.

Method descriptions specify their name and formal parameters, as usual, pre-conditions and termination conditions, and a sentential representation of a process net that describes the procedure to be carried out. It is important to note that the pre-conditions in method definitions are much stricter than the preference conditions associated with them in tasks; whereas preference conditions help determine the most appropriate method to use in a situation but are not required to be satisfied, by definition a method cannot be invoked unless its pre-conditions are satisfied. For example, if it is raining, a company might not want to maneuver through a valley, preferring higher and drier ground, though it will if it has to. However, the company cannot maneuver at all unless its vehicles are operational and have fuel - two necessary pre-conditions. The syntax for methods is:

```
<METHOD> ::=
  (:METHOD <name> (<variables>*)
    [ (:PRE-COND <condition>) ]
    [ (:TERM-COND <condition>) ]
    (:PROCESS <process> ) )
```



The main emphasis of method definitions is on specification of the process. Processes can be conceptualized in terms of process networks, or (possibly cyclic) directed graphs, where the nodes can be sub-tasks or primitive operators and the arcs between them indicate sequentiality (see Figure 1). Special nodes can split single paths into multiple paths, representing decision points or parallelism.

[1] Termination has the interpretation of failure, by default, though successful terminations could easily be added.

Figure 1. This figure shows a representative process network, where the sub-task nodes are labeled with T1..T5, C is a conditional node where execution will branch depending on the evaluation of the condition, and P is a split node at which execution proceeds along two threads simultaneously.

To encode process nets in TRL, we use a recursive sub-language to represent process networks, including features such as sequentiality, parallelism, branching, and loops. This language is analogous to various dynamic logics and event logics [7,8] for expressing a combination of actions and their ordering in a plan or computation.

```
<process> ::=
  (:SEQUENCE <process>+ )
  | (:PARALLEL <process>+ )
  | (:BRANCH <condit> <process> [<process>])
  | (:WHILE <condition> <process> )
  | (:FORALL <condition> <process> )
  | (<task-or-operator-name> <value>* )
```

Ultimately, process descriptions ground out in either calls to other sub-tasks (which themselves may have methods with processes, and so on) or operators which can be executed atomically. The meaning of the various process types is as follows: `:SEQUENCE`s specify a list of sub-processes that must be done in sequence; `:PARALLEL` processes cause divergence in the process such that sub-processes can be carried out simultaneously. `:BRANCH`es give a condition to evaluate that, if true causes the first sub-process to be done, else the second (or nothing if it is not given). `:WHILE` expressions continue to execute a sub-process while the specified condition remains true. And `:FORALL` constructs evaluate a condition, retrieve all variable bindings that satisfy the condition, and then initiate execution of multiple instantiations of the sub-process, one for each case, in parallel. Each of these types of processes can be nested within the others to build up complex procedures. Here is an example of a method with a process description based on the process net shown in the Figure above:

```
(:Method M1 (?a)
   (:pre-cond (not (moving ?a)))
   (:term-cond false)
   (:Process
     (:Parallel
       (:Sequence T1 T2 T3)
         (:Branch (:Cond C) T4 T5) ) ) )
```

3. Agent Algorithms

Given a set of task and method descriptions in TRL, an interpreter is needed for an agent to use this knowledge to decide what to do at each time step. We assume that an agent is initially assigned a set of top-level tasks and goals (which translate into tasks). The agent must select appropriate methods to accomplish these tasks, further decompose any sub-tasks, and so on, until it reaches concrete operators it can execute. After an action is selected and executed, the agent must maintain the state of its own progress through the process network to ensure it will execute subsequent steps in the plan. Finally, as time passes, conditions may change, so the agent needs to continually re-assess what are the best things to do.

We describe the algorithm using the notion of a task-decomposition tree (see Figure 2) and its use in a simulation. In the first time step of the simulation, the tasks are expanded downward from the top-level in the following manner. For each task, a method is selected whose pre-conditions are satisfied in the current state and for which no other method is more preferred (has higher priority). Then the process net for this method is placed at the next level down in the tree underneath this task. Then the process net is analyzed as a graph to determine the set of "topologically-first" things to do (this could be the first sub-task in a `:SEQUENCE`, or all of the sub-tasks in a `:PARALLEL`, for example). Each such sub-task is then recursively expanded to create deeper levels in the tree, until operators are reached, forming leaves. Importantly, if, at any time, no method with satisfied pre-conditions can be found for a task, then back-tracking is invoked by requiring the parent task to select another method (if there are any; if not, then back-tracking propagates upward further).

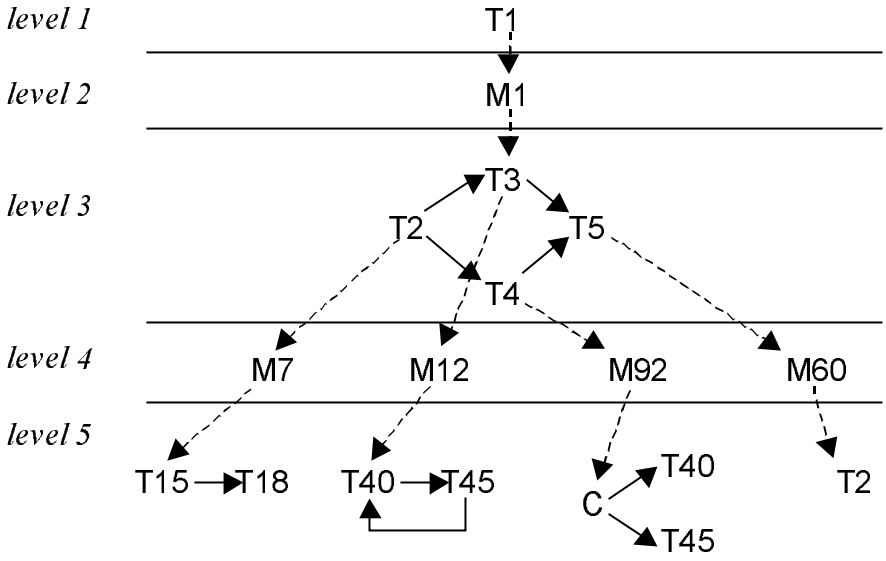


Figure 2. Task-decompostion tree. The levels alternate among task nodes and method nodes, with methods expanding into process networks (levels 3 and 5) that contain sub-tasks or primitive operations, and so on, producing the hierarchical structure. The arrows within a process net depict the execution of the net. For example, at level 3, the arrows from T2 to T3 and T4 indicate a parallel structure that merges again at T5. The dotted arrows to the methods indicate the methods selected to implement the task. Each task points to a unique selected method. The pointers from methods into the process nets keep track of which step (task node) they are at in the process, and these pointers get updated by the appropriate Step algorithm described below.

The algorithm for the agent is based upon recursion and the use of multiple threads for the parallel sub-tasks. The agent creates an instance of each task it is to execute. It then repeatedly calls a *Step* method of the task at each simulation time step until it is done or fails. Pseudo code for the top level agent routine is:

```
Agent(Task X)
// easily extensible to multiple tasks, or goals
   T = ConstructTask(X)
   do  // while notYet done or fail
      done = Step(T)
      // Step() will return one of three values:
      //     fail, notYet, or true
   until done=fail or done = true
end Agent
```

The agent will loop on each simulation step, repeatedly calling *Step* on the task until it is done or fails.

The construction of a task simply creates an environment for parameters, saves input arguments and initializes the method being used to null. The actual work of selecting methods to use is done in its *Step* routine.

```
ConstructTask(task T)
   create reference N for task T
   create an environment in which to hold variables
      for T and record any input parameters
   let A(T) be the set of all methods associated with task T
   // those methods with failing pre-conditions will
   //     be deleted in Step()
   current_method = null; // none selected yet
   return N
end ConstructTask
```

The real work at the task level occurs in the *Step* algorithm, which selects a method when necessary, constructs it, and invokes another version of *Step* on it.

```
Step(Task T)
// This is called on each simulation time step
   If (terminal_condition = true) return fail
   let A be set of possible methods A(T)
   rank the methods in A according to the 2-level
      preference/priority scheme
      // Note that preference conditions might change
      //     between simulation steps, resulting in sort
      //     order change
```

```
  let M be the first method in A (in sorted order)
  if M != current_method
    remove current_method from A
    // Removed to prevent flip-flopping of methods
    //    on successive steps
    for successive methods Mi in A (in sorted order)
      C = ConstructMethod(Mi)
      // returns reference to Mi or fail (due to
      //    failing pre-condition)
      if C=fail, remove Mi from A
      else select Mi and set current_method = Mi
      until method selected or A is empty
    if A is empty, return fail
  endif
  done = Step(C) // returns: fail, netYet, or true
  if done=fail
    // back-track by calling self to try another method
    remove current_method from A
    done = Step(T) // this is the recursive call
    // if fail is returned here, there are no more methods
    //    so failure propagates to higher level
  endif
  return done
end Step
```

Methods follow a similar paradigm. The construction simply sets up the method of execution of calls to *Step*. The real work at the method level is done in the *Step* routine.

```
ConstructMethod(Method M)
  create reference N for method M
  if ((pre_condition != true) or
    (terminal_condition = true)) return fail
  create an environment E in which to hold variables
    for M and record any input parameters
  let P be the process net for method M
  topologically sort the nodes in the network P
  let current_nodes(M) be the set of "first" nodes
    in P without predecessors
  // May have >1 element due to PARALLEL
  return N
end ConstructMethod
```

Like tasks, a method also has a *Step* function that is called on every simulation time step. It is within *Step* that new threads are created to accommodate the branches in a `:PARALLEL` structure. Throughout the execution of a method, the set *current_nodes(M)* will contain the parallel nodes that are ready to execute.

```
Step(Method M)
  let C be current_nodes(M)
  for each node Fj in C
    if Fj does not yet have a node reference Gj
      Gj = ConstructNode(Fj, E)
    If Gj = fail, return fail
  endif
  for all Fj in C do // in parallel
    donej = step(Gj)
    if donej = true
      remove Fj from C
      add successor(s) of Fj, if any, to C
    end if
  end for all
  if any donej = fail, return fail
  if all donej = true and C empty, return true
  else return notYet
end Step
```

At this point, the nodes that are to be constructed will be primitives that are either sub-tasks or operators. The former will lead to recursive construction and *Step* calls, while the latter will post the operations in a scheduler for execution. The appropriate constructor and *Step* routines are:

```
ConstructNode(Node N, E)
  if N is a sub-task
    R = ConstructTask(N)
    // using E to pick up any parameters to be
    //    sent to the task constructor
    Return R
  else
    if (pre_condition != true) return fail
    create reference R for node N
  endif
end ConstructNode
```

The result of this construction is that, if the node was a sub-task, the reference returned will be directly to the constructed task. Only if the node is an operator will the returned reference be to the operator[2]. Then, the *Step* method only has to handle operators.

```
Step(Operator N)
  done = post(N)
  // This call will be caught by the scheduler and
  //    not returned until some operation has
  //    been performed. The return will be either
  //    true or notYet.
  return done
end step
```

The result of executing the top level *Step* in the agent is that all operators it could execute in the current simulation time step (appearing as terminal nodes in the tree) are posted to the scheduler. The scheduler then selects one (at random - since any preference in ordering should have

[2] The actual implementation is slightly different, but this view makes the algorithm easier to understand.

been encoded explicitly using `:SEQUENCE`, etc.) and schedules it for execution. (The action itself may take some time, but initiating it is fast.) The results of the simulation step are encoded in the *done* value returned. That is used in the nodes above the operator to determine whether to schedule the same operator again or determine its successor. In subsequent steps, nodes representing actions that have been taken are deleted and replaced with their successor nodes in the corresponding process nets. When all the nodes in a process net at a given level have been completed, the parent task is marked as done and it is eventually deleted and replaced with its successors in the process net at the next level up, and so on.

```
Post (Operator N) // to the scheduler
// Note that this does not return to the invoking
//    thread until an operator has been executed
   do // forever
      wait until all operators have been posted
      randomly select one for execution and execute it
      return true for the operator selected
      return notYet for all other operators
   end do
end Post
```

There is one important aspect of this scheme that gives the agent the ability to react to real-time changes occurring in the environment around it.: the choices made in constructing the tree are re-verified at each time step. If conditions have changed, then some tasks or methods may no longer be valid or necessary, i.e. if a termination condition has become true. Therefore, with each time step the tree is traversed from top to bottom, checking for any problems; this is reflected in the *done* values passed all the way back up to the agent on each simulation step. If a termination condition has become true, then the sub-tree from that node down is deleted, and back-tracking commences (as in the initial stage) to try to find an alternative method for accomplishing the parent tasks higher up in the tree. We also note that, in this version, a method can get interrupted if a better method becomes available; this is not always the best thing to do, but the algorithms can be modified to make whatever choice is optimal for the domain.

4. Implementation

We have implemented a prototype version of a battlestaff simulation system based on this agent architecture in Java. The overall system design is shown in Figure 3. In the middle are the agent algorithms, running as Java threads. Below this is an interface to OTB. Our system currently is able to automatically monitor network traffic and extract information about entity states and positions from DIS PDUs. However, because of limitations of time, we have relied on a human ("puckster") to key in commands to OTB via a console, based on decisions made by our agent. Above the agent, the interface to brigade staff trainees is shown. This allows the brigade trainees to make commands and submit FRAGOs, make queries, and interact in other ways with the battalions. The agent itself makes use of two resources. First, all of the knowledge encoded in TRL is read in from a text file and "parsed" into Java objects representing each task and method definition. As the simulation runs, tasks and methods with specific input values get instantiated, and this spawns "live" versions of these objects which get linked into the task-decomposition tree by the agent. Finally, the agent also requires a mechanism for reasoning about the battlefield domain. For this, we use Jess (Java Expert System Shell), which is a Java implementation of a forward-chaining production system that reads CLIPS rules [9]. Jess contains all the state information the agents know about enemy and friendly entities, some of which is automatically updated from OTB, and other information is gained from communications with brigade, etc. In fact, a great deal of knowledge is also encoded in the Jess knowledge base for making inferences from the ground facts, such as degree of threat by enemy positions, which complements the procedural knowledge in TRL. Conditions in TRL expressions are evaluated by the agent by submitting them to Jess as queries, and new variable bindings are automatically extracted from the answers that are returned. The system is implemented in a distributed way using Java RMI to decouple the various modules, from agents to interfaces to inference engines, so that they can run on different machines.

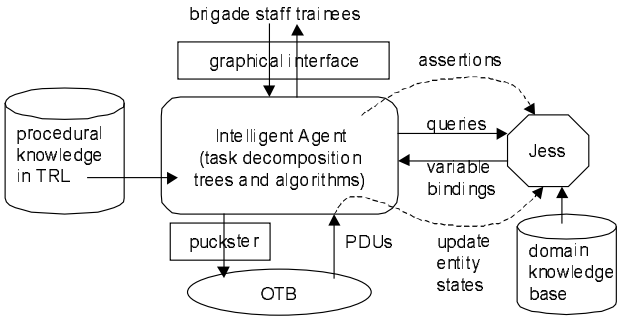


Figure 3. System Architecture.

5. Movement-to-Contact Scenario

To evaluate the capabilities of our agent-based simulation approach, we have developed a knowledge base specific for movement-to-contact (MTC) scenarios. MTC has several salient features that make it especially appropriate for our experiments. First, we focus on maneuver

because it involves interesting control and coordination issues, and develops progressively over time. Second, MTC also involves a large amount of information gathering and exchange, requiring communication between brigade and battalion. And third, MTCs are usually well-planned out before commencement. However, units must still be prepared to react in case there are unexpected surprises, since enemy movements are by definition unknown *a priori*.

In an MTC, the tasks that must be performed for various BFAs (battle functional areas) in a TOC are relatively well-defined and role assignments are clear. The S2 (intelligence officer) must report on enemy positions discovered by scouts. The scouts are typically instructed to fire and destroy enemy reconnaissance teams as soon as they are identified. Overall, the goal is to identify the movements and intentions of the OPFOR. The FSO (fire support officer) is supposed to coordinate the use of indirect fire to impede the progress of enemy main bodies. Finally, once the main bodies are identified, reserves can be committed, and friendly units must be positioned to meet and engage the OPFOR (preferably on favorable terrain), which is usually a brigade-level decision.

Developing a knowledge base for MTC has required us to go through a phase of knowledge acquisition. We have spent a great deal of time examining field manuals (e.g. FM 101-5 [10]), staff officers training guides, METLs, task analyses, etc., as well as talking with experienced military personnel. We have captured a preliminary form of this knowledge as a combination of task/method descriptions in TRL, and inference rules in Jess. To date, we have compiled 30 tasks descriptions, 45 method descriptions, 17 operators, and 185 rules.

We have been exploring the range of behaviors that can be produced by our system on various hypothetical tactical situations on the terrain in the Ft. Hood military base. We have obtained a terrain database that can be loaded into OTB. We also pre-load our agent knowledge base with initial routes for battalions and companies, observation posts (OPs), targeted areas of interest (TAIs), etc., derived from operation orders (OPORDs). When the simulation starts, the enemy and friendly units are set in motion in OTB. Our system automatically tracks entity positions and status by reading entity state PDUs off of the network. As enemy units come into sight, spot reports are generated by our battalion agents and sent to the brigade interface. They appear in a GUI where staff officers trainees can read them. The battalion agents may also submit requests for support or calls for fire to brigade. Finally, the battalion agents may output messages to a "puckster" instructing him/her to move entities directly in OTB or to make them to fire on targets (this could be completely automated in the future). Importantly, the battalions will automatically stop and defend themselves as necessary if enemy appear in unexpected or threatening positions. In addition to receiving reports from the battalions, the brigade staff officers may take actions through the GUI, such as submitting requests for information (RFIs) or status reports. As the information accumulates and the situation clarifies, the brigade commander can issue orders, e.g. to commit reserves or re-direct subordinate units (via FRAGOs). The MTC scenario is over when the situational picture is complete and the decision has been made to engage the OPFOR with specific units at specific locations.

6. Discussion

Our approach to agent design can be used in other domains, especially where methods and procedures for carrying out tasks are well-defined. This is the case for many functional areas in a TOC, as documented in TTP, etc. In other domains, such as fire-fighting or flight-control, similar task knowledge can be extracted from training manuals or domain experts. This knowledge can be encoded in TRL, and then the algorithms we presented can be used to make appropriate decisions for agents based on this knowledge. However, it useful to point out that this approach differs significantly from many other approaches to intelligent-agent design. Many of these other approaches are based explicitly on a goal-regression or a more general planning architecture [11,12]. Such methods require knowledge of the actions the agent can take, and then specifically take advantage of an explicit specification of post-conditions or effects of actions to select sequences of actions that will achieve their goals [13]. This is much harder to express in a domain where tasks are hierarchical because effects of high-level tasks are a complex combination of the effects of their lower-level sub-tasks, which can be difficult to infer. Furthermore, in our domain of interest - battlefields - there are many actions going on simultaneously, and many actions (such as sending a message or shooting at a target) have only probabilistic outcomes anyway. Fortunately, training of military personnel is largely doctrine-based, and methods for carrying out tasks have for the most part already been defined. Hence to select a method to achieve a task, all one has to do is look up the pre-enumerated list of possible methods, and then evaluate which is the best to use based on some known preference criteria. No explicit calculation of effects needs to be performed, which is probably a benefit for soldiers in the heat of a battle. Our agent algorithms are designed to simply follow these task definitions in a similar way. There could be some benefit to being able to reason more flexibly with the procedural knowledge to solve a wider range of (potentially unanticipated)

problems, but this is not possible in the current framework.

One limitation of TRL is that it does not have any built-in mechanism for dealing with temporal aspects of decision making. It is easy to imagine situations where the preference between two methods for the same task depends on how much time is available to complete the task, or an attack should not be performed until 5 minutes after the enemy has entered a given zone. Furthermore, it would probably be useful to encode some indication of how long various procedures take (though they might be iterative or indefinite), and also to make it easy to express things like timeout conditions on operators. Currently, there is no syntactic way of encoding such information in TRL, and the algorithms do not support the use of such information. However, the inquiry system for Jess recognizes a special variable ?curr_time and returns a simulation time. This can be combined with times associated with events entered from the OTB interface to provide a limited mechanism for dealing with time. Developing a more general way of dealing with time is a potential area of future work; we plan on studying extensions to TRL with explicit temporal aspects. Computationally, this might be handled by using a constraint-propagation algorithm to select a sequence of actions that meets a deadline or satisfies other types of constraints [14,15]. However, uncertainty is also anticipated to play a significant role, so methods for making estimates from probability distributions over time are likely to be needed.

7. Conclusion

In this paper, we have presented the intelligent agent architecture underlying our work on the University XXI project. The objective was to provide more realistic behaviors of aggregate-level units in battlefield simulations. In particular we were interested in simulating the cooperative actions of battalions, which drives much of the interaction and information-exchange in a tactical operation center. We also wanted to simulate the more complex decision-making of such units, where they must follow orders under nominal conditions, yet react appropriately to situations as they arise. To accomplish these goals, we developed a knowledge representation language called TRL for capturing typical procedures that staff officers go through to carry out their various functions. Then we presented algorithms for an agent to use this knowledge to select appropriate actions at each time step, using a task-decomposition tree to keep track of multiple activities going on in parallel, and also to modify and repair this tree when situational conditions change. This general approach could be used to simulate the intelligent behaviors of a wide variety of units in military simulations. Within our University XXI project, much additional work remains. Our ultimate goal is to use these agent-based simulations to support the training of brigade TOC staff members, which requires both the extension of our TRL knowledge-base to handle a broader range of operations beyond MTC, and also to enhance the brigade interfaces to provide additional interactions and communications with their battalions and other units.

References

[1] DoD 5000.59-P (1995). DoD Modeling and Simulation (M&S) Master Plan. `http://www.dmso.mil/documents/policy/msmp/index.html`

[2] Nwana, H.S. (1996). Software Agents: An Overview. *Knowledge Engineering Review*, 11(3):1-40.

[3] Fikes, R. E. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*,5(2):189-208.

[4] Sacerdoti, E.D. (1977). *A Structure for Plans and Behavior*. Elsevier: New York.

[5] Wilkins, D.E. (1984). Domain-Independent Planning: Representation and Plan Generation. *Artificial Intelligence*, 22:269-301.

[6] Ingrand, F.F., Chatila, R., Alami, R., and Robert, R. (1996). PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. *Proceedings of IEEE International Conference on Robotics and Automation*.

[7] Stuart, C.J. (1987). Branching Regular Expressions and Multi-Agent Plans. in *Reasoning About Actions and Plans*. M. Georgeff and A. Lansky (eds.). Morgan Kaufmann: San Mateo, CA..

[8] Davis, E. (1990). *Representations of Common Sense Knowledge*, Morgan Kaufmann: San Mateo, CA. pp. 225-229.

[9] Friedman-Hill, E. (1997). *Jess, the Java Expert System Shell*. Sandia National Laboratories, Livermore,CA. `http://herzberg.ca.sandia.gov/`

[10] Field Manual No. FM 101-5 (1995). *Staff Organization and Operations*. Department of the Army, Washington, D.C.

[11] Laird, J., Newell, A. and Rosenbloom, P. (1987). SOAR: An Architecture for General Intelligence. *Artificial Intelligence*, 33:1-33.

[12] Jones, R.J., Laird, J.E., Nielsen, P.E., Coulter, J., Kenny, P., and Koss, F.V. (1999). Automated Intelligent Pilots for Combat Flight Simulation. *AI Magazine*, 20(1):27-41.

[13] Wooldridge, M. and Jennings, N. (1995). Intelligent Agents: Theory and Practice, *Knowledge Engineering Review*, 10(2):115-152.

[14] Allen, J.F. (1991). Temporal Reasoning and Planning. in *Reasoning About Plans*. J.F. Allen, H.A. Kautz, R.N. Pelavin, and J.D. Tenenberg (eds.). Morgan Kaufmann: San Mateo, CA.

[15] Dean, T., Firby, R.J., and Miller, D. (1988). Hierarchical Planning Involving Deadline, Travel Time, and Resources. *Computational Intelligence*, 4(4):381-398.

Acknowledgements

The authors would like to thank Dr. James Wall and Randy Elms in the Texas Center for Applied Technology on the Texas A&M campus for valuable input on this project. We would also like to thank the team of graduate students who worked on implementing the system: Vijayavani Nori, Keith Biggers, Karthik Morapakkam Josium, Sumitha Bhandarkar, Neeraj Khandelwal, Yu Zhang, and Mark Watson. Finally, we would like to acknowledge the cooperation of other people involved in the larger scope of the University XXI project, including personnel at Ft. Hood and University of Texas at Austin. We especially thank Dr. Bruce Porter for helpful feedback on technical details.

Author Biographies

THOMAS R. IOERGER is an Assistant Professor in the Department of Computer Science at Texas A&M University. He received his B.S. degree in Molecular and Cellular Biology in 1989 from the Pennsylvania State University, and his M.S. and Ph.D. degrees in Computer Science in 1992 and 1996, respectively, from the University of Illinois, where he held an NSF Graduate Fellowship. His research interests are in the areas of artificial intelligence, machine learning, and intelligent agents. He also conducts research on applications of AI and pattern recognition to problems in computational biology.

RICHARD A. VOLZ, an IEEE Fellow, is Royce E. Wisenbaker Professor of Engineering in the Computer Science Department at Texas A&M University. He served as Department Head from 1988 to 1997, and has recently embarked upon a new initiative on "Training Systems Science and Technology." Prior to joining Texas A&M, Dr. Volz was Director the Robotics Research Laboratory and Professor of Electrical Engineering and Computer Science at the University of Michigan. He has served on five federal advisory boards: 1) the Air Force Scientific Advisory Board, 2) the Ada Board, 3) The Aerospace Safety Advisory Panel, a Congressional oversight committee on NASA, 4) the NASA Space Station Advisory Panel, and 5) the NASA Center of Excellence in Information Technology Advisory Panel. He has received the Decoration for Exceptional Civilian Service from the U.S. Air Force, two NASA Special Service Awards and the IEEE Millenium Medal. He has served as the general chairman of the 1990 IEEE International Conference on Robotics and Automation, as Editor-in-Chief of the IEEE Transactions on Robotics and Automation from 1994-99, as well in as numerous other editorial and conference organization positions. He is the author or co-author of over 100 research papers and has led over $12,000,000 in funded research projects. Dr. Volz received his B.S., M.S., and Ph.D. degrees in Electrical Engineering from Northwestern University in 1960, 1961 and 1964 respectively.

JOHN YEN received his B.S. in Electrical Engineering from National Taiwan University, Taipei, Taiwan in 1980, his M.S. in Computer Science from University of Santa Clara, CA in 1982, and his Ph.D. in Computer Science from the University of California, Berkeley in 1986. Dr. John Yen is currently a Professor of Computer Science and the Director of Center for Fuzzy Logic, Robotics, and Intelligent Systems at Texas A&M University. Before joining Texas A&M in 1989, he had been conducting AI research as a Research Scientist at Information Sciences Institute of University of Southern California. His research interests include intelligent agents, fuzzy logic, and evolutionary computing. He has published more than 100 technical papers in journals, conference proceedings, and edited volumes. Dr. Yen is the Vice President for Publication of IEEE Neural Networks Council. He is a member of Editorial Board of several international journals on fuzzy logic and intelligent systems. He co-authored the textbook (with R. Langari) *Fuzzy Logic: Intelligence, Control, and Information* published by Prentice Hall in 1999. Dr. Yen received an NSF Young Investigator Award in 1992 and he is a Fellow of IEEE.