Livak, T., Heffernan, N. T., Moyer, D. (2004). Using Cognitive Models for Computer Generated Forces and Human. Tutoring. 13th Annual Conference on (BRIMS) Behavior Representation in Modeling and Simulation. Simulation Interoperability Standards Organization. Arlington, VA. Summer 2004

Using Cognitive Models for Computer Generated Forces and Human Tutoring

Tom Livak
Neil Heffernan
Dale Moyer
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609
508-831-5569
tomlivak@cs.wpi.edu, nth@cs.wpi.edu

Keywords:
Cognitive model, Computer Generated Forces, Human Tutoring

ABSTRACT: *The computer generated forces community and the online training community do not share much overlap. The small overlap that currently exists is that training groups need to use computer generated forces, but these two tasks are implemented separately and in different ways. This paper presents a method to unify these two seemingly disparate areas, by using a single cognitive model to provide both tutoring and computer generated forces capability. We have built a prototype system that uses this technique to deliver both computer generated forces and tutoring to multiple human players in a 3D first-person simulation.*

1 Introduction

In cooperation with the US Army we are working on a system to tutor soldiers on the use of military operations on urban terrain (MOUT). One goal of the system is to tutor a whole platoon, which is a group of 30 soldiers, while they complete a military exercise. Each soldier would be sitting at a computer running a 3D simulation of the exercise, controlling a virtual soldier, or avatar, that can interact with both the environment as well as the other soldiers. The system will give the soldiers feedback and advice as they proceed through the exercise. This paper discusses some of the difficulties in building such a system, and a conceptual architecture for dealing with them. We have built a prototype system to demonstrate the architecture.

There are many 3D simulation systems out there, so one goal of this project is to develop a system that can be used with one of them to tutor human soldiers, as opposed to developing our own simulation. This system will need to communicate with the simulation, as well determine what feedback to give to the soldiers and when. Another goal is that the system is able to adapt to different simulations.

Another feature of the system is that it should have the ability to use computer generated forces (CGFs) as simulated teammates. Given that the exercise may involve up to 30 men, we may not have 30 soldiers to participate in the exercise. Therefore we want to have CGFs take the place of the real soldiers in the exercise.

2 The Problem

We have two components in our system that need to be modeled. Modeling is a difficult task, and so we want to limit the amount of it we need to do. The first component is the CGFs. These computer controlled soldiers need to act like real soldiers, or else they will not be effective to train with. In addition, we want to them to act as if they were human; that is, they should make mistakes. Soldiers will have to learn how deal with errors that are made in the field. Therefore our CGFs should attempt to be as close to simulating how an actual soldier could respond to a given situation. There has been much research in developing CGFs for MOUT tasks, for instance Best and Lebiere [1], however we need our CGFs to match the skills we are trying to tutor, so we can not use their research directly. Laird has done work developing intelligent opponents for MOUT training [2], but we are looking for computer generated teammates.

The second component is used to tutor human soldiers. By tutor we mean to give appropriate feedback to soldiers based on their performance in the exercise. When a soldier does well, the system should give positive feedback; likewise if they fail or make mistakes the system should give negative feedback. In addition to feedback, the system must provide assistance to soldiers who do not know what to do. For the feedback to be useful however, the system must be aware of the soldier's current context. MOUT tactics used by soldiers are complex, with many variations. Given a situation, there are many correct things to do. Therefore we must know the state of the student's mind at the time in order to give the appropriate feedback. In order to do that we must have a model of the student. We can then use model tracing algorithms to figure out the reasoning behind their actions, and give

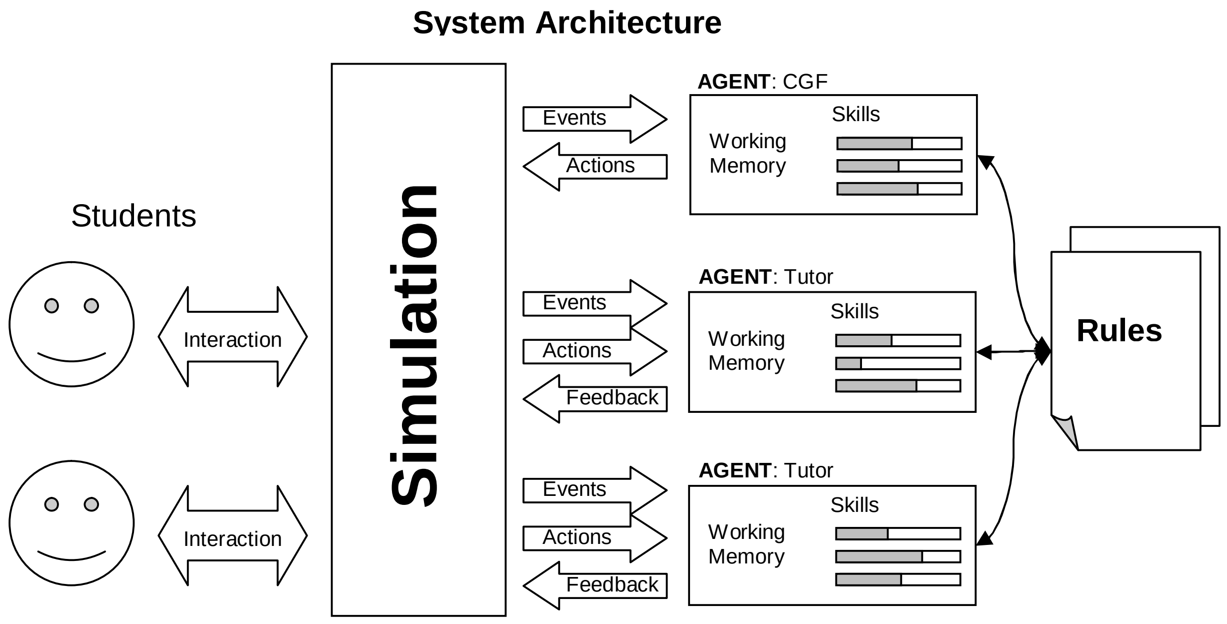

Figure 1: Conceptual Architecture

the student appropriate feedback and assistance. Model tracing is a plan recognition algorithm, which we describe below. Model tracing tutors [3, 4] have been used by many and have been shown to be effective [5].

3 The Solution

Our solution is to realize that our two modeling tasks are one in the same. The computer controlled forces are attempting to model a real soldier. For the tutoring task, we need a model of a real soldier to give appropriate feedback. They both use a model of a real soldier; however they use it in different ways. The computer controlled forces use the model to produce actions from a given situation. For the tutoring task we take the given situation and the user's actions to determine a line of reasoning, which is then used to give feedback. Therefore we can use forward chaining for the computer controlled forces, and backward chaining for the tutoring task. We can then develop a single model of production rules that can be used for both tasks. For this project, we are only exploring rule-based cognitive models.

Figure 1 shows the overall architecture of our system. On the left side the students interact with the simulation. They will be producing some input, either via a keyboard, joystick, or other input device, to control their avatars in the simulation. The simulation will produce some output the students can perceive, typically a graphical display and audio effects.

On the right of the diagram is the set of production rules that form the cognitive model. Connected to the rules are what we refer to as the agents. Every agent is an instance of the cognitive model, and is connected to an avatar in the simulation. All the agents use the same set of rules, but each has a distinct working memory, which represents the knowledge and goals that a particular agent has. The working memory of different agents will be different, since each will perceive different events in the world. Additionally, they start with a different working memory, for instance the squad leader knows he is the squad leader, and has a different set of goals and knowledge than does the platoon leader. There are two types of agents, one for CGFs and one for human tutoring. CGF agents command their avatars in the simulation; the human tutoring agents provide feedback to students.

Each agent also has a distinct set of skill levels. A skill level represents the probability that the student knows that particular skill. Skills can either be a single rule or a group of rules. Skill levels could be set for the computer generated forces, so that they would sometimes make mistakes. These skill levels are determined by a process called knowledge tracing. The technique is not described completely here but it is described in Corbett and Anderson [6].

The agents are connected to the simulation over some communication channel. There are three types of messages that are exchanged: events, actions, and feedback. Events are things that happened in the simulation that the soldiers can perceive. Examples would be hearing footsteps, seeing enemy soldiers, receiving orders from a team leader, etc. These are always sent from the simulation to the agents. The simulation is responsible for deciding which agents receive the events. For instance, if the event is seeing enemy soldiers, it is up to the simulation to decide which soldiers can see the enemies in the simulation. The agents use events to update their working memory.

Actions are things that the soldiers can do. Examples include moving, shooting, giving orders, etc. Where actions are sent depends on the mode of the agent. CGFs send actions to the simulation; their avatars then perform the actions in the simulation. Tutoring agents receive actions; these are the actions the students took. The tutoring agent then uses a model tracing algorithm, described below, to provide feedback.

There is an important distinction between what we refer to as instant actions and latent actions. The difference between them is that instant actions are executed in the simulation atomically, where a latent action happens over time. This is important because it means that we can only detect latent actions when the action is complete, not when the actions starts. For instance, when a player moves from one place to another, we cannot tell when he starts to move where his final destination is going to be. Generally latent actions have an event that occurs when the action is complete. For instance, when an avatar moves, an event is generated that specifies where the avatar currently is. In general the model will use these events to determine when its current action is finished and can move on to the next. When we are doing human tutoring, we can look at these completion events and infer the actions the students have made. This is important because we want to capture that the student's action was to move to a particular place, not just to move.

Feedback messages are used to provide feedback to the student. They are only sent from the tutoring agents to the simulation. The simulation relays the feedback to the student. Examples of feedback include displaying text of the screen, or highlighting an area of interest on a map.

In order to illustrate how to two types of agents use the set of rules, we present Table 1, which shows a sample of six rules that could be used in our system. We have not implemented these rules yet, but use them to illustrate the architecture because they show how the communication between the soldiers works. For purposes of illustration, these rules are vastly simplified from the actual rules that we would use; in addition, they are English language versions of rules we would normally code in a computer language. These rules have four components. The first is type, which marks the rule as either a correct rule or an incorrect one. Incorrect rules are generally typical mistakes soldier might make; they are used when tutoring human soldiers. The next two parts are the "if" and "then" clauses of the rule; when the conditions under the if clauses are true, then the actions under the then clause should fire. Finally there is a message, which is used to give feedback. Because the rules are presented with English language if-then clauses, the message field appears to be the same as the "then" clause. However in an actual implementation the then clause would be encoded in a programming language.

Rule 1	
type	correct
if	your goal is to clear a building AND there is a room with a threat
then	order team to clear that room [action]
message	"Don't bypass threats; clear a threatened room."

Rule 2	
type	correct
if	your goal is to clear a building AND there is a room without a threat AND there is no room with a threat
then	order team to clear that room [action]
message	"Order the team to clear a room."

Rule 3	
type	incorrect
if	your goal is to clear a building AND there is a room without a threat AND there is a room with a threat
then	order team to clear the unthreatened room [action]
message	"There is a room that contains a threat, you should have cleared that room first."

Rule 4	
type	correct
if	You see an enemy enter a room AND You are not the team leader
then	tell leader you saw an enemy [action]
message	"You need to tell the leader you saw an enemy enter a room."

Rule 5	
type	correct
if	You see an enemy enter a room
then	consider that room threatened [change memory]
message	"You saw an enemy enter a room; you must consider that room threatened"

Rule 6	
type	correct
if	you are told an enemy entered a room
then	consider that room threatened [change memory]
message	"You were told an enemy entered a room; you must consider that room threatened"

Table 1: Sample Set of Rules

Four of the six rules have actions as their consequence, and two change working memory. There is no restriction that rules must do one or the other, in practice, most rules that produce an action also change working memory. However the distinction between rules that produce actions and those that do not will be important when we discuss the model tracing algorithm below.

3.1 Computer generated forces

When used to run computer generated forces, we apply the rule using forward chaining. Consider an example where several things are currently in the agent's working memory: you are the team leader, the goal is to clear a building, there are three rooms that you can reach, you saw an enemy enter a room, and a team member told you he saw an enemy enter a different room. Rules 5 and 6 would fire, which would cause working memory to be updated, so that there are now two threatened rooms and one unthreatened room. Rule 3 could now fire, except that it is an incorrect action and we assume, for the moment, that the computer generated forces do not make mistakes. Rule 1 will fire, however there are two rooms that the agent could choose to clear. There are no other rules that give preference one way or the other, so the system will arbitrarily chose one. The action produced by Rule 1 will then be sent to the simulation so that his team will be ordered to clear a room.

3.2 Human tutoring

Now let us consider what happens when the system is used to tutor a human soldier. Let us take the scenario above, with the same things in working memory. There are three actions the human soldier could take, each action being to order his team to clear one of the three rooms. Two of these rooms should be considered threatened, and clearing either room is only correct action. Should the soldier order his team to clear a room, we can use backward chaining to determine that it is correct action. We see that the action could be the result of Rule 1, 2 or 3. To see which one could fire, the system needs to determine if the room selected is threatened or not. Rule 5 and 6 can fire to show that the room is threatened, which show that Rule 1 could fire, and so we found a set of rules that from working memory produce the desired action. Since Rule 1 is a correct rule, we know the student has made a correct action. It is important to notice it does not matter which of two threatened rooms the human soldier choose to clear, they are both considered correct. This is an important aspect of the model tracing; the student is given the flexibility to solve the problem as they see fit.

If the student had chosen to clear the unthreatened room, then a different set of rules would be traced. Again, Rules 1, 2 and 3 could all lead to the action chosen, but after considering Rules 5 and 6 and the state of working memory, Rule 3 is found to be the source of the action. Rule 3 is marked as an incorrect action, so the system needs to tell the student they have made a mistake. It can easily do this by displaying the message associated with the rule to the student. In this case the student would be told "There is a room that contains a threat, you should have cleared that room first."

The system can give additional information about the mistake by displaying the messages of the rules that caused the room to be considered threat; in this case the student would see "You were told an enemy entered a room; you must consider that room threatened" or "You saw an enemy enter a room; you must consider that room threatened." The student therefore gets immediate feedback that is appropriate to the situation.

If the student doesn't know which room to clear, he can ask the system for assistance. The system can run the system in forward chaining mode to determine what one of the correct actions would be. This will is done in same way as when the CGF ran the model; Rule 5 and 6 will fire causing Rule 1 to fire. However this time the system keeps track of the messages that are associated with these rules, and then presents them to the student. It presents each message in the order they were fired until the student makes an action. In this case the system would produce "You saw an enemy enter a room; you must consider that room threatened" followed by "Don't bypass threats; clear a threatened room." By presenting each message individually, the system will only give enough assistance as the student needs.

4 Implementation

We have built a prototype system that implements the above architecture. The cognitive model used is not complex, but the purpose is to show it is possible to use the same model for both the CGFs and for tutoring students. A screenshot of our system can be seen in Figure 2.

We are using Unreal Tournament 2003 (UT2003), a commercial off the shelf game, as our simulation system. UT2003 allows the users to make modifications to the game to support different game types. These modifications are written in a language called UnrealScript. We have written such a modification that has several responsibilities. First, it maintains a TCP/IP connection to a server program we have written that we call UTJess. These programs communicate using a protocol we developed which is described below. The modification also detects events and student actions and sends them to UTJess program. It also receives actions for the CGFs, and has the avatars perform those actions in the simulation. Finally, it takes the feedback messages from the UTJess program and relays them to student in an approriate manner. The UTJess program is a simple program that has three main purposes: to translate network messages to and from working memory, running the models for-

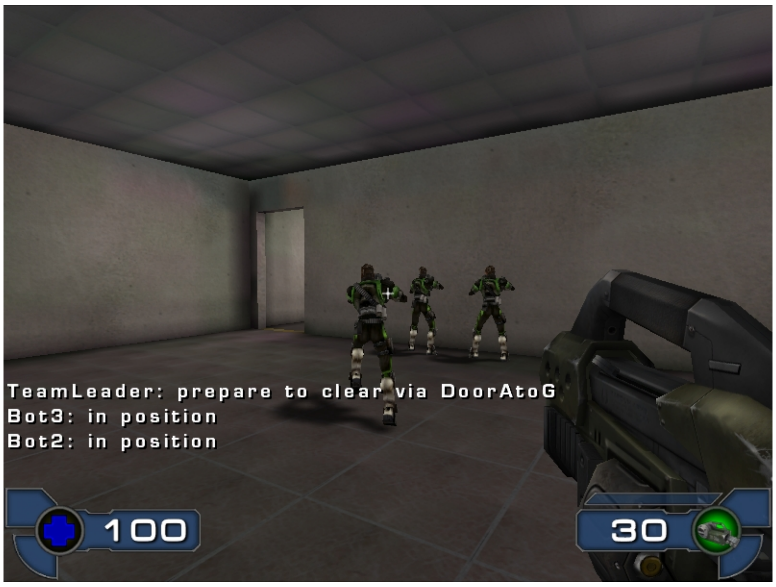
TeamLeader: prepare to clear via DoorAtoG
Bot3: in position
Bot2: in position

Figure 2: Screenshot of our prototype system.

ward for CGFs, and implementing the model tracing and knowledge tracing algorithms for tutoring.

4.1 The environment

The tutoring system should be able to support different scenarios. Each scenario will have different features, such as buildings, trees, enemy forces and so on. Also each scenario will have different objectives. Each scenario could also be defined by events that happen during the exercise, for instance we could have a scenario that deals with a platoon leader being killed in combat. We however define a scenario only on its initial settings. Currently events that happen during a scenario can not be forced, nor can they be suppressed. So during any exercise the platoon leader could be killed in combat, and the soldiers will need to act accordingly.

UT2003 supports custom maps, as well as placing custom information in the maps. Our system stores all the scenario information in a map file. In addition to information about the objectives, we store other information in the map. One problem with computer controlled forces is that they are blind. A human player can look at a room in UT2003 and identify where all the exits are, but it is not as easy for the computer to do. Therefore we placed information about the location of rooms and doors into the map that allow the computer to tell where things are. Another way that we use extra information to help the computer controlled forces is by predefining paths. The MOUT doctrine has specific rules about how rooms should be entered, that involve such factors the form of the room and whether the doors swing in or out. In order to simplify this for the computer controlled forces, we place all of the paths in the map. This information is also used when tutoring human players, to make sure they are moving along the correct paths.

In this sense, the model has more information than the student does. We do this because it is a very difficult problem to model perception, so instead of trying to determine whether or not the student has seen, for instance, all the doors in a given room, we assume that he has. For this domain, this is generally not a problem, because the student has to make sure that they are fully aware of their situation. However, this can lead to suboptimal tutoring. For instance, if a student is in a room, and doesn't see any more rooms to clear, they may backtrack. This is an error, and the system will give them a diagnostic message "You need to clear all rooms before backtracking." The student will likely look back and find the room they were supposed to clear, but the system would have been more helpful if it had said "You did not see a room, you need to clear it before backtracking." However this can only be done if we model student perception very finely. For our prototype, we have not done this, but we hope to model some elements of perception more accurately in the future.

4.2 The cognitive model

JESS (Java Expert System Shell) was used to implement the cognitive model as a series of production rules[1]. We currently have 24 rules that code a simple model of clearing a building. These rules are categorized over six different goals: clearing a building, clearing a room, moving, shooting, waiting, and controlling civilians. Each goal can have subgoals; in this way clearing a building is composed of several clear room goals. Clearing rooms is in turn composed of movement goals and wait goals. A movement goal represents the task of moving along a particular path, while a wait goal represents the need for the

1. We were previously using a simplified version of ACT-R[3], but moved to JESS because it is well documented and has greater cross-platform capabilites

team to wait until everyone is ready before moving on. The controlling civilians goal represents the task of securing and watching over civilians. Finally, the shooting goal is fairly simple: shoot enemies, do not shoot civilians or teammates. The shooting goal is implied, in that it is assumed that every soldier always has that goal. The other goals represent tasks that are started and completed.

The clearing building goal is represented by four rules. The first rule *BeginClearBuilding*[2] recognizes that the soldier has been ordered to clear a building, and creates the goal. The *ClearNextRoom* rule recognizes that there is another room to clear, and orders the team to clear that room. The *BackTrack* rule fires when there are no rooms to clear, that is, the team has reached a dead-end, and orders the team backtrack. The final rule *FinishClearBuilding* determines when the entire building is clear, and marks the goal as complete and retracts it.

Clearing rooms is represented by five rules. The first rule, *BeginClearRoom*, recognizes that the soldier has been ordered to clear a room and creates the goal. The *Stack* rule is fired first, which moves the soldier to stack outside the doorway by setting a move goal. A wait goal is also set so the soldier waits until the rest of the team is stacked at the doorway. When they are all ready, the *Assault* rule can fire, which sets a move goal to move the soldier into the room as prescribed by the MOUT doctrine. Once they are in the room, the *TakeCommand* rule can fire, but only for the team leader. This rule recognizes that there are civilians in the room, and the consequence of the rule is that the team leader orders the civilians to clear out of the room. Another team will have the control civilians goal, and will watch over the civilians. The last rule *FinishClearRoom* fires when all enemies and civilians in the room are dealt with, and completes and retracts the goal.

The remaining rules deal with moving, waiting, shooting and watching over civilians. They are fairly straight forward so we do not discuss all of them in detail here, but show two JESS rules used by our system in Table 2 so that we can see how these rules are implemented. These rules are both part of the implied shooting goal. The rules are very similar, and both check some knowledge in working memory, and produce an action. However, the *ShootCivilian* rule is an incorrect rule, that is, it should never fire for a properly behaving student. It is marked by the (incorrect) token so that CGFs do not run it. The two rules also have messages associated with them; *ShootEnemy* has a hint message that is shown when the rule is applicable and the student asks for help. The *ShootCivilian* rule has a buggy message that is displayed to the student if he uses the rule, that is, this is the message that provides negative feedback.

```
(defrule ShootEnemy "Engaging enemy"
  ; figure out which room we’re in
  (self (room ?room))

  ; is there a enemy in this room?
  (person
    (name ?person)
    (type enemy)
    (room ?room)
  )
=>
  ; hint message
  (assert (advice-message
    (message "You need to engage the enemy")
  ) )

  ; produce the action
  (assert (shoot-person-action
    (person ?person)
  ) )
)

(defrule ShootCivilian "Violating ROE"
  ; mark this as an incorrect action
  (incorrect)

  ; figure out which room we’re in
  (self (room ?room))

  ; is there a civilian in this room?
  (person
    (name ?person)
    (type civilian)
    (room ?room)
  )
=>
  ; bug message
  (assert (advice-message
    (message "BUG: Do not shoot civilians!")
  ) )

  ; produce the action
  (assert (shoot-person-action
    (person ?person)
  ) )
)
```

Table 2: Two rules from the prototype architecture.

4.3 Algorithms

We can implement model tracing using backward chaining. JESS has a facility for doing backward chaining,

2. We will write all rule names in italics

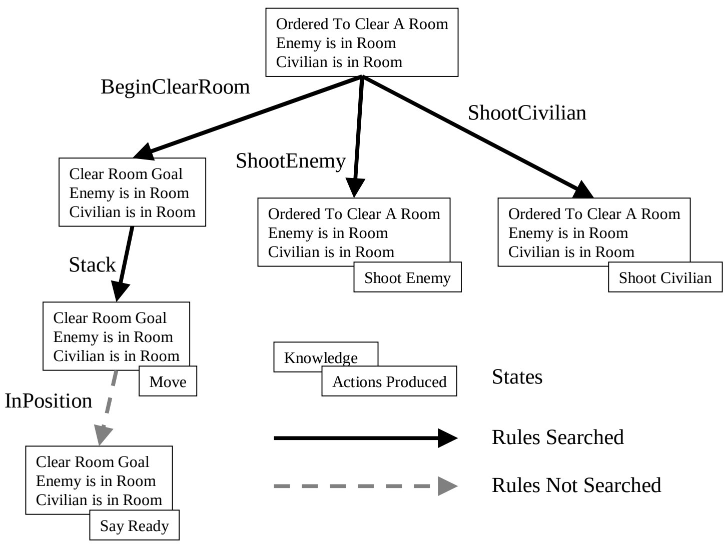

Figure 3: Portion of a model tracing search

however it is primarily a forward chaining system and its backward chaining capabilities are limited. There are not many systems that can do both forward and backward chaining, so we implemented model tracing in a way that does not rely on backward chaining.

To implement the model tracing algorithm, we used a depth-first search. Whenever we need to trace the model, we start with current knowledge base and then look at rule that can fire. We fire one of these rules, and again look at rules that can fire. When we have tried all the rules at a certain level, we backtrack. When a rule produces an action, we stop and backtrack. We continue until we have exhausted the search. In this way we can find all the possible actions from a given knowledge state. Part of an example search is shown in Figure 3. Here we see that from the inital state, three rules can fire. *BeginClearRoom* can fire because the soldier has been ordered clear a room, and the soldier can shoot either civilans or enemies because they are present. We are concerned only with the possible actions the soldier can take, not whether or not they are correct. When *BeginClearRoom* fires, it modifies working memory so that there is *ClearRoomGoal*, which allows the *Stack* rule to fire. This produces a movement action. This is simplified from the actual implementation, where there are several movement rules so that the soldier follows a particular path. Once the movement is complete, the *IsPosition* rule can fire and the soldier should tell his teammates that he's ready. However, this rule will not be searched because the previous state produced an action. Therefore, the set of possible actions contains three elements: shooting a civilian, shooting an enemy, and moving to a stack position. Whenever the student takes one of these actions we must recompute the set of possible actions. Additionally, whenever an event occurs we must recompute the set of possible actions because the event will change the current state of working memory.

Since we used a depth-first search, it is possible that our system could get into an infinite loop. However the model is simple, and the domain does not lend itself to rules that could produce looping. However, other, more general systems have used an iterative deeping search to prevent infinite looping [7]. Also, more general model tracing algorithms assume that if a given action is not in the set of possible actions, then the student has made an error. We have modified this assumption slightly to take into account latent actions. For instance, if the student walks forward a few feet, we do not want to call that an error unless the model specifically says that movement is an error. Therefore, when a latent action is not traced, we ignore it, and do not produce an error message.

The distinction between rules that produce actions and rules that do comes in to play here. We can only observe the actions that students perform; therefore we can not directly tell whether or not the student has performed the rules that do not produce actions. For instance, in Figure 3, we can not tell whether or not the student has recognized that they have been ordered to clear a room, that is, performed the *BeginClearRoom* rule, if they have not taken any actions. However, we can infer that they have if they then move into a stacking position. This is why we must search through all the rules until find a sequence that produces an action. The strength of the model tracing algorithm is that we can detect these rules, and therefore track whether or not a student knows them using knowledge tracing.

Our system implements Corbett and Anderson's knowledge tracing algorithm[6] to determine the skill levels of the students. Unlike most model tracing tutors, we also track how often the student has made common errors. For each rule we can attach a name. Whenever a rule

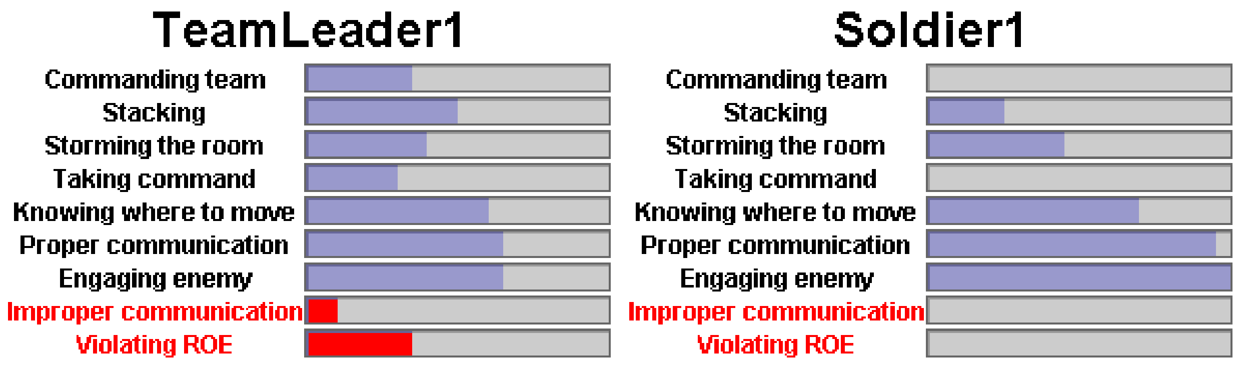

Figure 4: Knowledge tracing skill bars

is fired, we can increase the probability that the student has learned that rule. Likewise, when the rule is not fired when it should be, we can decrease that probability of knowing that rule. For the two rules in Table 2, the skill names are “Engaging Enemy” and “Violating ROE[3].” Using the knowledge of what skills the student knows, and what the student needs to “unlearn,” we can grade the aptitude of the student. These skills are represented to the user or instructor by skill bars, which can be seen in Figure 4. “Violating ROE” and “Improper Communication” are both errors, and appear in red. Although our knowledge tracing algorithm is completely implemented, for each skill we should also have several parameters which represent how hard it is to learn or forget a skill. Since we have not completed our model we have not been able to do any testing to determine these parameters properly.

4.4 Network Protocol

We have a simple network protocol. Each message is an ASCII string of the form [header, timestamp, param1, param2]. A list of all the messages we use can be found in Table 3.

There are four classes of messages: setup, actions, events, and feedback messages. Actions, events, and feedback messages are those described in the conceptual architecture. Setup messages are used to prepare the exercise. The information about the scenario, including the information about doors, rooms, etc, are all sent from the simulation to the agent at the beginning of the exercise. This information is stored in working memory. The simulation also sends information about the participants in the exercise, such as the rank of the individual and which teams they are part of. Additionally a message is sent to the agent to tell it which soldier it is controlling.

5 Evaluation

The purpose of our prototype is to determine whether or not the conceptual architecture we developed is feasible. In order to test that we built a small scenario for a fireteam (four soldiers) consisting of a series of connected rooms. The exercise is to clear all the rooms properly. The purpose of this evaluation was not to judge the effectiveness of our CGFs or the effectiveness of the system at teaching students; we are only trying to show that that our architecture is sound.

We ran the exercise with four CGFs, and they cleared all the rooms in the proper manner, that is, by stacking outside of the door, waiting until the team was ready, then assaulting the room as a group, and finally moving into a formation inside the room.

We also ran the same exercise with two CGFs, and one student playing the role of the team leader and another playing the role of another team member. The team leader could choose to clear the rooms in any order they chose, and the system would give the student feedback saying that they had made a correct decision. Either student could also ask for hints at any point in the exercise, and the system would give them appropriate feedback. For instance, when a student was clearing a room, the system gave these messages:

> “your goal is to clear a room”
> “you need to move into position”
> “move to the highlighted node”

After the last message the system would also tell UT2003 to display a graphic on top of the position the student needed to move to. The system also gives feedback when the students make a mistake. Once a student moved into position, he should tell his teammates he is in position so that they know when everyone is ready to enter the room. If the student said he was in position before moving to the correct location the system would respond with “incorrect action: you said you were in position, but you are not in position.” Also the CGFs acted as they should; they followed the team leader’s orders and stacked outside rooms, and told their team members they were in position as before.

3. Rules of Engagement

Setup Messages	
Path	Describes a path
Door	Describes a door
Room	Describes a room
Person	Describes a person
Team	Describes a team
Self	Tells the agent which person it's controlling

Event Messages	
At	Avatar is at a certain location
Receive Ready Heard	Someone said they were in position
Receive Clear Room	Was ordered to clear a room, through a particular door
Receive Clear Building	Was ordered to clear the building
Person Died	Some person died
Person Changed Room	Some person changed rooms
Person Changed Type	Some person changed type

Action Messages	
Move To	Move the avatar to particular location
Say Ready	Tell your team you are ready
Say Clear Room	Tell your team to clear a room through a particular door
Say Get Down	Tell civilians to get down
Say Move Out	Tell civilians to move out
Shoot Person	Shoot a person

Feedback Messages	
Advice	Displays some text on the users screen
Highlight	Displays a graphic image over a particular location

Table 3: Communication Protocol

6 Future Work

This paper represents the beginnings of this project. There are many things that we will continue to work on in the future. At this point, our model is very simple, but demostrates the functionality of our conceptual architecture. Work is currently being done to extend the model to handle more of the MOUT doctrine.

One aspect that we have not dealt with much is the runtime performance of the system. Given that the model tracing is using a search, it is possible that with a large number of rules that the performance would be unacceptable. However, this would only affect the tutoring, as the CGF simply use forward chaining. The system is designed to distributed, with the simulation and the cognitive model running on seperate machines. However, the system currently runs reasonably well when they are both running on the same machine.

One simple optimization we have made is to use a lazy approach to model tracing. As stated earlier, whenever an event occurs we need to model trace and recompute the set of possible actions. However, we only need this set when the student has performed an action. If several events occur before the student makes an action, we will waste time recomputing the set. Instead we mark the set as invalid, and the next time we need it we recompute it. This is different than recomputing the graph everytime an action is performed, because some actions, such as movement, can be ignored. For instance if the set of possible actions contains only a communication action, we do not want to recompute the set for every step of movement the student makes.

We also employed some heuristics to increase the search speed. We know that some of the rules are independent and can be applied in any order. For instance, when hearing two teammate say they are ready, if does not matter in which order the rules fired to mark them as ready. Even though the order does not matter, the naïve search will search both orders. We mark such rules and when searching, we only consider one order. There is much work that can be done in finding more heuristics to speed up the search.

The purpose of using one model for two tasks is that, presumably, less effort goes into developing the system overall. However, if it takes as long to make a model that does two tasks as it does to make two models for seperate tasks, nothing has been saved. Since we developed our model from scratch to do both CGFs and tutoring, we cannot say for certain that any work was saved, although we feel that the effort was less than if we had built two models independently. However we believe that it is not necessary to build the model from scratch in order to use it for both tasks.

To test this belief, we are currently looking for an existing model of CGFs that we can convert into a tutoring model. Although we do not have a model yet, looking at some other models shows promising results. For instance, Laird's SOAR Quakebot[8] has an architecture similar to ours, where the simulation sends events to the model and the model responds with actions. This gives us hope that it may be possible to add the tutoring part of our architecture. Table 4 has a production from TacAir-Soar with an accompanying source comment. Most of TacAir-Soar's productions are similar to this one, in that they look at the current knowledge of the world, and then create new goals or execute actions, much like the rules in our cognitive model. In order for this particular rule to be used in model tracing, we would have to attach a skill name and a hint message to this production. The skill name could be "Employ Weapons", and the hint message could be "There is a bandit out there, and you need to

```
; Propose employ-weapons if there is a
; bandit out there, but not if we should
; be doing something more important, like
; chasing him, confusing him, bugging out,
; or evading a missile.

(sp intercept*suggest-proposal*employ-weapons
 (goal <g> ^problem-space.name intercept
           ^state <s>)
 (<s> ^bogey <b>)
 (<b> ^roe-achieved *yes*
      ^intention known-hostile
      ^contact *yes*
      ^intercept-geometry-selected *yes*)
 -{ (goal <g> ^operator <o> +)
    (<o> ^name << pincer chase-bandit
                  change-piece-of-sky
                  bug-out evade blow-through
                  blow-through-continue >>) }
-->
 (<s> ^suggest-proposal <p> + &)
 (<p> ^name employ-weapons ^bogey <b>)
)
```

Table 4: A production from TacAir-Soar[9]

employ weapons." Of course we can not be sure that this is all that is needed to make TacAir-Soar into a tutor until we actually try to implement it. Seeing that TacAir-Soar has over 5,000 rules, we plan to start with a smaller model first.

7 Conclusion

The purpose of this paper was to show that one could use the same cognitive model for two tasks, computer generated forces and human tutoring. Our prototype system shows that this is indeed possible. In addition we have developed a conceptual architecture that generalizes this approach to using a cognitive model in this manner. The potential benefits of this approach are clear: the necessary modeling for such a task is cut in half. More practically, however, is that there has already been much research on development of computer generated forces. We hope that by using the techniques in this paper, it would not be hard to extend these some of these existing models into models that can also be used as tutors.

8 Acknowledgements

This work was partially funded by the Office of Naval Research.

References

[1] B. Best, C. Lebiere, and C. Scarpinatto. A model of synthetic opponents in MOUT training simulations using the ACT-R cognitive architecture. In *Eleventh Conference on Computer Generated Forces and Behavior Representation*, Orlando, Florida, 2002.

[2] Robert R. Wray, John E. Laird, Andrew Nuxoll, and Randolph M. Jones. Intelligent opponents for virtual reality trainers. In *Interservice/Industry Training, Simulation and Education Conference (I/ITSEC) 2002*, Orlando, Florida, December 2002.

[3] J. R. Anderson and R. Pelletier. A developmental system for model-tracing tutors. In Lawrence Birnbaum, editor, *The International Conference on the Learning Sciences. Association for the Advancement of Computing in Education*, pages 1–8, Charlottesville, Virginia, 1991.

[4] K. VanLehn, R. Freedman, P. Jordan, C. Murray, R. Osan, Ringenberg M., C. Rose, K. Schulze, R. Shelby, D. Treacy, A. Weinstein, and M. Wintersgill. Fading and deepening: The next steps for andes and other model-tracing tutors. In G. Gauthier, C. Frasson, and K. VanLehn, editors, *Proceedings of the 5th International Conference on Intelligent Tutoring Systems*, pages 474–483, Montreal, Canada, 2000. New York: Springer.

[5] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier. Cognitive tutors: lessons learned. *The Journal of the Learning Sciences*, 4(2):167–207, 1995.

[6] A. Corbett and J. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction*, 1995.

[7] K. Koedinger, V. Aleven, and N. Heffernan. Tools towards reducing the costs of designing, building, and testing cognitive models. In *2003 Conference on Behavior Representation in Modeling and Simulation*, 2003.

[8] John E. Laird. An exploration into computer games and computer generated forces. In *The Eighth Conference on Computer Generated Forces and Behavior Representation*, Orlando, Florida, May 2000.

[9] Soar IFOR project. `http://www.isi.edu/soar/soar-ifor-project.html`.

Author Biographies

TOM LIVAK is a master's student at Worcester Polytechnic Institute in Massachusetts. He is working on a thesis in the area of collaborative tutoring systems. He received his BS in Computer Science and Mathematics from Worcester Polytechnic Institute.

NEIL HEFFERNAN is a professor of Computer Science at Worcester Polytechnic Institute in Massachusetts researching intelligent agents, tutoring systems, and machine learning. He received his PhD in Computer Science from Carnegie Mellon.

DALE MOYER is a recent graduate of Worcester Polytechnic Institute, receiving his BS in Computer Science.