Collaborative Warrior Tutoring

by

Thomas M. Livak

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

By

Date: August 2004

APPROVED:

Professor Neil Heffernan, Thesis Advisor

Professor George Heineman, Thesis Reader

Professor Michael Gennert, Head of Department

Contents

List of Figures

List of Tables

Abstract

Much work has been done to develop intelligent tutoring systems in domains such as algebra, geometry, and computer programming. Our work is to develop an intelligent tutoring system to train US soldiers. One main difference in this domain is that one of the main skills to be learned is cooperation between teammates, so the tutor must emphasize collaboration as a skill. In addition, to help train this skill the system must be able to run in real-time, and provide both computer generated teammates, as well as intelligent opposing forces. This system is the first real-time, multi-user, model tracing tutor with simulated teammates. The goal of this thesis is to build a prototype system to validate that this is a valid approach for this domain.

1 Introduction

In cooperation with the US Army we are attempting to answer the question "how can we combine intelligent tutoring systems with military simulations?" To that end we have developed a system to tutor soldiers on the use of military operations on urban terrain (MOUT). The system is designed to tutor a group of soldiers while they complete a military exercise. Each soldier will 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 thesis discusses some of the difficulties in building such a system, and a conceptual architecture for dealing with them.

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 a large number of soldiers, we may not have enough students to participate in the exercise. Therefore we want to have CGFs take the place of the real soldiers in the exercise. Having CGFs is important because we want our system to be able to teach students how to collaborate with their teammates.

The system should also be flexible, in two ways. First, if there are multiple ways to solve a problem, we want to allow the student to solve it in the manner they see fit. Therefore the system must be flexible in allowing multiple solutions to problems. Second, during an operation, events may occur that require the soldiers to change what they are doing. Therefore the system must be flexible to allow for changing goals during the operation.

2 Background

There are two main fields of research that this project involves. First is the research on building intelligent tutoring systems. These have historically been for tutoring only a single student, although there is some work on collaborative tutoring. The second field is research on building military simulations for training purposes.

2.1 Intelligent Tutoring

The earliest systems that used computers to provide tutoring to students were known as Computer-Aided Instruction (CAI). These systems would ask the student a series of questions, and would tell the student whether their answer was right or wrong. These systems can give feedback on why certain answers

are wrong or give broad hints on how to solve problems, but they can not give more detailed feedback. In particular they cannot give feedback that is directed towards a particular student.

The next step were systems we call Intelligent Tutoring Systems (ITS). These systems could give feedback on the individual steps that would led to the solution. Additionally, because the systems could track where the student was in the problem, they could give hints that were tailored to the individual student's progress. Likewise, when the student made a mistake, the feedback messages given can be more appropriate. The hope was that these systems would be more effective at tutoring than CAI systems.

Bloom did a study that compared than effects of human tutoring versus conventional classroom study[1]. A control group received conventional classroom instruction with about 30 students per teacher. The experimental group received one on one tutoring. At the end of the experiments both groups were tested. Bloom found a effect size of two sigma, which means the average score of the experimental group was two standard deviations above the average score of the control group. This means that 98% of those in the experimental group did better than the average score of the control group, which is a huge effect. Two sigma has become the goal that intelligent tutoring systems have strived to achieve. Kulik and Kulik[2] did a study that involved a meta-analysis of several CAI systems, and they found an effect size of 0.4 sigma. Koedinger et al[3] found in a study of their model tracing tutor that their tutor had an effect size of one sigma. Likewise, the Practical Algebra Tutor (PAT) was shown to have an effect size of about one sigma for the skills it was designed to teach.[4] PAT also had an effect size 0.3 on two standardized tests: the Iowa Algebra Aptitude and a subset of the Math SAT suitable for ninth graders. We can see that ITS systems can perform much better than CAI systems, but there is still room for improvement.

2.1.1 Approaches to Tutoring

There are two approaches to tutoring. They are not so much discrete, different approaches as they are opposite ends of the spectrum. The first method is discovery learning, where you give the students the means to learn and allow them to explore the problem space. The hope is that with the right tools they will learn how the solve the problem on their own. Some argue that discovery learning is better for teaching ideas and principles, because the student will better remember what they discover themselves.

A good example of a discovery learning system is Green Globs[5]. In this system the student is presented with a set of coordinate axes with several green globs distributed randomly. The student is asked to input a function, such that function crosses as many globs as possible. The idea is that in trying to find functions that hit more globs they will determine the shapes and properties of linear, quadratic, and other functions.

On the other end of the spectrum is coached problem solving. In coached problem solving the student is given a task, and while they perform the task

they are given feedback. This feedback will eventually direct the student to the solution. This process is repeated until the student learns the task. Some argue that coached problem solving can be more appropriate then discovery learning, because in many cases discovering how to do the task can be difficult and frustrating.

One approach to coached problem solving is constraint-based tutoring[6, 7]. In constraint-based tutoring one builds a set of constraints on the problem state. The student is allowed to work on the problem, trying to reach some goal state. If they ever violate any of the constraints, appropriate feedback is given.

Another approach is model tracing[8, 9]. To build a model tracing tutor one must first build a cognitive model that knows how to perform the task to be tutored. As the student works on the problem, the student's actions are compared to the actions the model produces to determine if they are correct or not. One advantage of model tracing is that because the model "knows" how to perform the task, it can give the student hints on how to proceed. However, compared to constraint-based tutoring, it is more difficult to build a model tracing tutor.

Another advantage of model tracing tutors is that they can be used to track the student's knowledge of the individual skills needed. The most popular method is known as knowledge tracing[10]. First, every rule in the model is associated with a skill, although this is not necessarily a one-to-one mapping. Every skill is assumed to be in a learned or unlearned state, and once the skill is learned, it can not be unlearned. Four parameters are attached to each skill

$p(L_0)$ **Initial** the probability the skill is learned prior to the first opportunity to apply the skill

$p(T)$ **Transition** the probability that the student's knowledge will transition to a learned state after an opportunity to apply the skill

$p(G)$ **Guess** the probability that the student will guess correctly from an unlearned state

$p(S)$ **Slip** the probability that the student will make a mistake from a learned state

These parameters need to empirically estimated for each skill. Given these parameters we can estimate the probability $p(L_n)$ that the student has learned the skill after n attempts by a Bayesian inference process. Though it is fairly simple technique, knowledge tracing is fairly effective. Some research is being done on more complex methods of tracking knowledge[11].

2.1.2 Cognitive Architectures

In order to build a model tracing tutor, we need to build a cognitive model. A cognitive architecture provides a framework to develop a cognitive model. Most cognitive architectures are production rules systems, and all are based on some theory of cognition. Some examples of cognitive architectures include SOAR[12],

ACT-R[13], and COGNET[14]. Each models different aspects of cognition, for instance ACT-R models the limited capacity of short term memory and the amount of time it takes to transfer information from long term memory to short term memory.

An alternative to using a cognitive architecture is to build a cognitive model using a general purpose production rule system. The advantage of this is that building the model will most likely be simpler, however the disadvantage is that the model will not benefit from the features of cognition simulated by the architecture. However, in many applications, these features may not necessary. For instance, it most tutoring applications we are not concerned with modeling the capacity of short term memory.

All production rule systems store information in working memory, and have a set of rules. These rules have some conditions on the contents of working memory, and some actions to change working memory. When the conditions for a rule are met, the rule fires and the actions are taken. Production rule systems such as OPS5, ART, CLIPS, and JESS, are forward chaining systems, and given an initial state of working memory fire any rules that apply. They use the Rete algorithm[15] to quickly match elements from working memory to conditions. The Rete algorithm takes advantage of the fact that most rule sets will contain rules that share conditions. By only evaluating these shared conditions once, the Rete algorithm can quickly determine which rules are ready to fire. Other systems, such as Prolog or MYCIN, use backward chaining. These systems take a goal state of working memory, and working backwards, find a sequence of rules and an initial state of working memory that could lead to the goal state.[16]

2.1.3 Collaboration Research

There has been research on collaboration in intelligent tutoring systems. Algebra Jam[17] is a system that incorporates collaboration by allowing several students to work on an algebra problem together. It includes several tools to enhance collaboration, for instance an object oriented chat. Each student is sitting at their own computer, but while using the text based chat can highlight parts of the screen to show the other students what they are talking about. What makes Algebra Jam novel, however, is that they try to track the different roles that students are taking, for instance, a leadership role, a critic role, or an observer role. The hope is by understanding how the student is collaborating with the other students the system can understand their actions better. For instance most tutoring system interpret non-activity as a lack of knowledge. However Algebra Jam may be able to make an more informed decision if it knows the student is acting as an observer.

Another system that uses collaboration is the Web-based Haskell Adaptive Tutor (WHAT)[18], which uses collaboration to teach the programming language Haskell. WHAT includes similar tools for collaboration as Algebra Jam, however WHAT allows for simulated virtual students. The real students are not told which of their collaborators are virtual. The virtual students are not perfect, in fact they designed to make mistakes. The idea is that by helping the

virtual students, the real students will learn more.

Both of these systems are interesting, however, like many other systems in field of collaborative tutoring, they differ from our system in that they are using collaboration to teach, not teaching collaboration. Neither programming Haskell or solving algebra requires collaboration, but the MOUT doctrine requires teamwork.

2.2 Military Simulations

The second major area of research this project involves is military simulations. There are many different simulation systems. All provide some simulation of the physical world, and some are more detailed than others. Additionally, computer generated forces (CGFs) may be simulated. CGFs may either be opposing forces or teammates. CGFs are often their own systems which are integrated with a simulation system.

2.2.1 Simulation Systems

Many simulations are built using the High Level Architecture (HLA)[19]. HLA defines a protocol that different simulations can use to communicate. Simulations have been built for aircraft, tanks, and dismounted infantry, and they can all work together through the HLA. Although it's possible to extract some information from a simulation, the general nature of the HLA allows only very basic information to be distributed. This makes it impossible to develop a tutoring system solely through the HLA. Instead modifications must be made to the simulation system.

At Fort Benning's Dismounted Battle Space Lab, they are using the Soldier Visualization System (SVS)[20] for training. Each soldier is placed in a 10 foot room and a virtual environment is projected on to the walls. Each soldier has sensors placed on their helmet and gun so that the system can track whether they are standing or crouching and where they are pointing their gun. The simulation strives to be an immersive environment, however it does not have any intelligent tutoring capabilities. It is also a closed proprietary system, meaning changes can not be made to it to allow tutoring.

Many researchers are starting to use game technology for their simulation purposes[21]. These systems are cheaper, in both the cost of the system and the hardware needed to run them. Game engines are also generally built to flexible, and it is fairly easy to modify them for new purposes. Some of the games being used include America's Army, Half-Life, and Unreal.

2.2.2 Computer Generated Forces

There has been a lot of research in developing computer generated forces, both as teammates and as opposing forces. The SOAR IFOR (Intelligent FORces) project[22] is exploring the use of the SOAR cognitive architecture in developed simulated pilots, although eventually they hope to build CGFs for other

domains. Their TacAir-SOAR model has been used as opposing forces in a number of large scale military training operations.

In domain of MOUT operations, there have been a number of projects researching CGFs. For the Virtual Training & Environments (VIRTE) program there have been two projects of note. Wray et al[23] developed a SOAR model for opposing forces. The other project is Best et al's[24] work in building an ACT-R model of computer generated teammates. Both of these projects used Unreal as their simulation. While these projects illuminated some of the issues in developing CGFs, the models are not publicly available so we could not use their research directly.

2.2.3 Military Training

There has been research into how to build tutoring systems for military applications, but the field is quite varied. For instance, the Operator Machine Interface Assistant (OMIA)[25], is a system that tutors pilot on flying the MH-60S and MH-60R helicopters. These helicopters require collaboration between the pilot and copilot, and the system runs in real-time. However, OMIA differs from our system in that it uses a state-based tutoring system as opposed to model tracing. Their approach uses a finite state machine (FSM) to determine the possible correct actions at any point in time, and is similar to model tracing in that the students actions are compared to those produced by the FSM. While FSMs are easier to build, a new FSM must be built for each scenario, where as the cognitive model built for model tracing should be flexible to any situation.

The Conversational Agents in a Pattern Oriented Training Environment (CAPOTE)[26] system has explored the relation between the simulation and the tutoring system. CAPOTE is designed to teach pilots situational awareness, radio communications, and flight pattern geometry. Pilots study these skills and then practice them in flight simulators. However these flight simulators are expensive and the students do not get to spend a lot of time in them. Instead of building a complete flight simulator, CAPOTE simulates only what is really necessary to learn these skills. CAPOTE provides two dimensional graphics for the flight pattern geometry, and synthetic agents that provide conversational abilities for the radio communications. By finding the right amount of simulation needed for the task, CAPOTE can provide a more cost-effective training method.

Another tutoring system that is similar to ours is the Advanced Embedded Training System (AETS)[27]. AETS is used to tutor the air defense team for Navy ships. The task is real-time and requires collaboration between the teammates. AETS also uses a model tracing approach to provide tutoring. AETS differs from our system in that is an embedded system, that is, it runs on the actual machines used in Navy ships. Also, AETS focuses on very low level actions, such as tracking the eye movements of the students. A lot of the work that AETS does is translating these low level actions into more high level actions, such as looking at a part of the screen and recognizing a threat on radar.

3 Design

We have two components in our system that need to be cognitively modeled. Cognitive 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.

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 to give the appropriate feedback. 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 the student appropriate feedback and assistance.

Our solution is to realize that our two modeling tasks are one and 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.

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, because every soldier in our system is taking the role of an infantryman. In a more complex systems different agents may have different sets of rules. Each agent 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

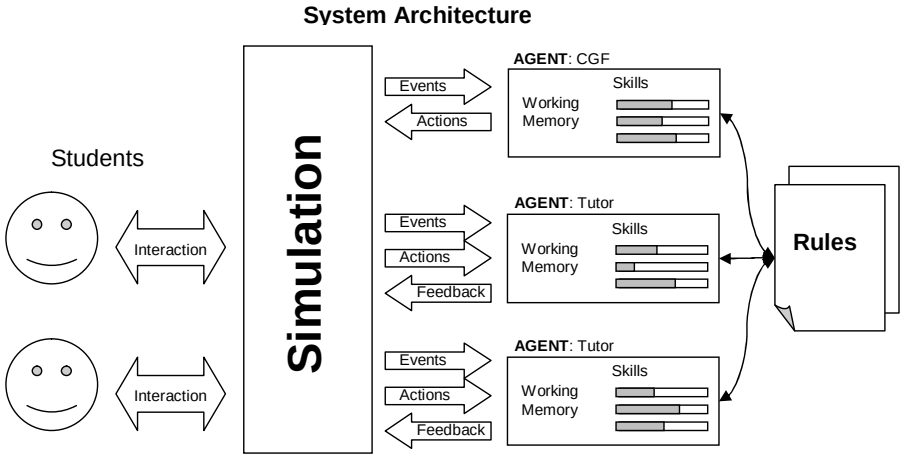

Figure 1: Conceptual Architecture

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, and ranges from 0 to 1. 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 knowledge tracing.

The agents are connected to the simulation over a 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 type of 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. Some actions become events for other soldiers. For instance, if the team leader orders his team to clear a room, that is an action,

but the rest of the team will receive an event telling them to clear the room.

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 the avatar's current location. In general the model will use these events to determine when its current action is finished and can move on to the next. For instance, when a CGF decides to move somewhere, it will issue a move action. The model will then wait for an event telling it that its current location is where it wants to be before it continues with its other goals. When we are doing human tutoring, the system can look at these completion events and infer the actions the students have made. When the student moves, a event is generated describing the avatar's location. We can infer there was a movement action, and model will trace both the rules that generated the action and the rules that waited for the action's completion. It should be noted that these location events are not based on cartesian coordinates, but rather on annotations on the map as described in the Implementation section. Determining latent actions after they have completed is important because the system should 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 the 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. These are not the rules we have implemented, but use them as simple examples to illustrate the architecture. For purposes of illustration, these rules are vastly simplified from the actual rules that we used; in addition, they are English language versions of rules we would normally code in a computer language. The complete set of rules we implemented can be found in Appendix A. The rules have four components. The first is type, which marks the rule as either a correct rule or an incorrect one. Incorrect rules are 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.

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,

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

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 choose one. Generally production rule systems will have a conflict resolution strategy to determine which rule to fire when many apply, but our system does not. This is because when two rules apply it represents that there are two approaches to take to the current situation. Therefore the system can fire either rule and the result is still correct. 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 threatened 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 chooses 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 be done in the 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 each time the student asks for help. 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, which the Unreal game interprets. 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 appropriate 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 forward for CGFs, and implementing the model tracing and knowledge tracing algorithms for tutoring. The UTJess program also determines which latent actions have been performed by looking at events, although moving is currently the only latent action.

Figure 2: Screenshot of our prototype system

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. We could define scenarios by the events that happen during the exercise. For instance, a scenario might say that the team leader will be shot after the clearing the third room. Our system does not do this, and defines a scenario only on its initial settings. So during any exercise the team leader may or may not be killed after clearing the third room, 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 model is supposed to be an expert student, and 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. The complete model can be found in Appendix A. We currently have 24 rules that code a simple model of clearing a building. Working memory is set of objects, each object belonging to a particular template. A template defines the attributes, or slot as they are called in JESS, that the objects of that template have. For instance, there is a *person* template, which has *name* slot. There may be two *person* objects in working memory, with *names* Joe and Bob. The templates we used can also be seen in Appendix A. We assume that working memory elements are never forgotten. This is not a problem because the model is supposed to know how to do the task correctly, so it should not forget things. Events are represented by working memory elements that are automatically asserted by the UTJess program. Likewise actions are working memory elements that automatically retracted and sent to Unreal by the UTJess program.

The production 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 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 *Begin-ClearBuilding* 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.

4.3 Algorithms

We can implement model tracing using backward chaining. JESS has a facility for doing backward chaining, 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. This is actually the typical approach to model tracing[28].

To implement the model tracing algorithm, we used a depth-first search. Whenever we need to trace the model, we start with current state of working and then look at rules 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

```
(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)
  )
=>
  ; buggy 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.

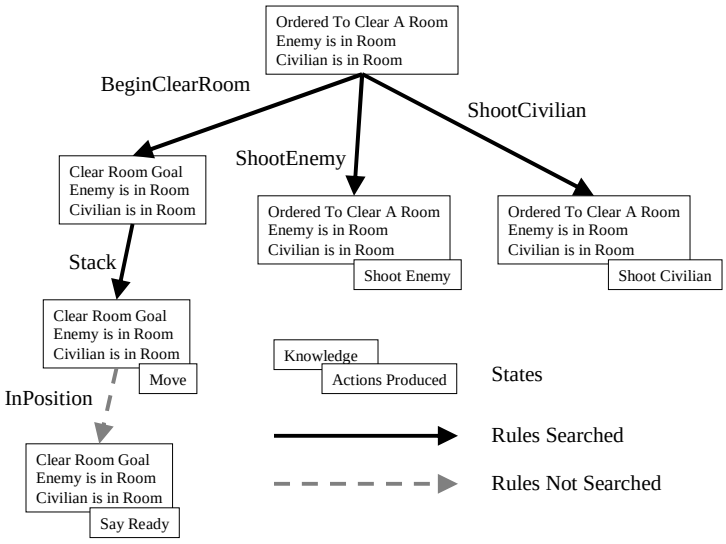

Figure 3: Portion of a model tracing search

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 initial state, three rules can fire. *BeginClearRoom* can fire because the soldier has been ordered clear a room, and the soldier can shoot either civilians 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.

One simple optimization we have made is to use a lazy approach to model tracing. 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 set every time 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 teammates 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.

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 deepening search to prevent infinite looping [29]. 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 we 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 the knowledge tracing algorithm 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 skill name. Whenever a rule is fired, we can update our estimate of whether a student knows the skill by a Bayesian inference process. For the two rules in Table 2, the skill names are "Engaging Enemy" and "Violating ROE[1]." 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

[1]Rules of Engagement

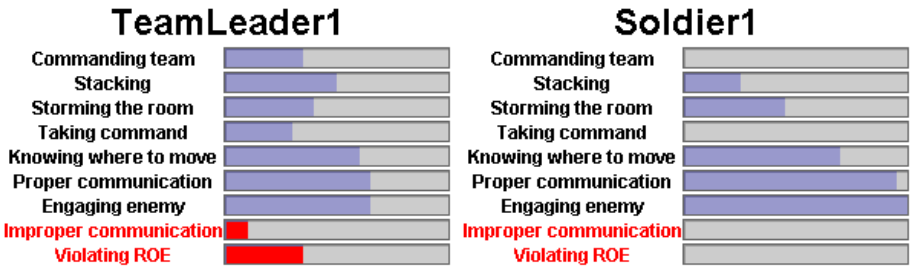

Figure 4: Knowledge tracing skill bars

knowledge tracing algorithm is completely implemented, for each skill we should need to empirically estimate the four parameters described in the Background section. However we do not have access to soldiers to test our system, so we can not determine these parameters.

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 fire team (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 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.

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

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.

Once concern we had was the run-time performance of the system. Given that the model tracing is using an exhaustive 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. This concern affects all model tracing tutors, not just our system. Some research has been done on how the branching factor and the depth of the search affects the model tracing time[28]. For our system we did a simple evaluation to determine how much collaboration was affecting the model tracing time. The concern is that having more people working together may cause the time spent model tracing to rise intractably. To test this we ran an experiment comparing the time spent model tracing to the size of the team. We had a human participant play the team leader with zero to three CGF teammates. We recorded the total time spent model tracing as the participant completed the task of clearing a single room. We only recorded the time that was spent model tracing, not the time spent processing other information, and we took the average over five runs to account for variability. The results can be seen in Figure 5. We see that there is a lot of variability, but the time required to model trace appears to constant. If this relation holds, then as more students are added to the simulation, then total time processing all the different students would increase linearly. The system is designed to be distributed, with the simulation and the cognitive model running on separate machines, and so if the linear response holds, the system can scale to any number of users by increasing the number of computer systems used. However, the time spent model tracing is very dependent on the exact model, so we can not be sure the linear response will hold for more complex models.

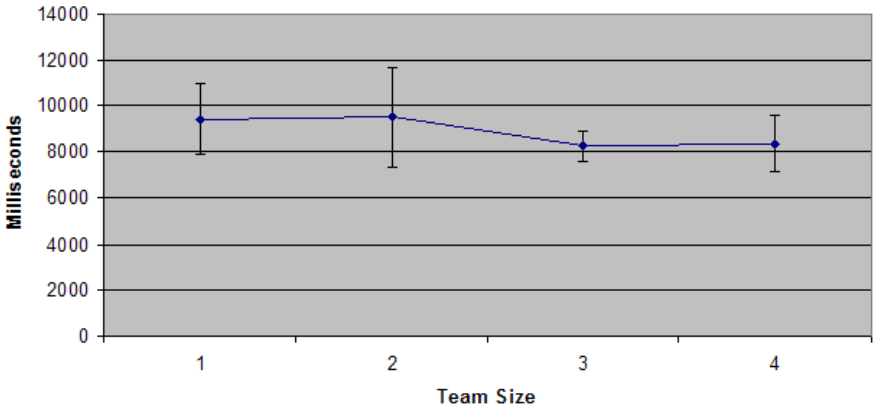

Figure 5: Team Size vs Model Tracing Time (97.5% confidence intervals)

6 Future Work

There are many things that could be explored for future research on this project. At this point, our model is very simple, but demonstrates the functionality of our conceptual architecture. Work could be done to extend the model to handle more of the MOUT doctrine.

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 separate 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, SOAR Quakebot[30] 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 employ

```
; 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[22]

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. However, we were not able to get in contact with the authors of any models that were willing to participate.

Currently our system is using a single model that produces all tutoring. However, a better method involves the use of two models[31], a student model and a tutor model. The student model provides a cognitive model of the student; our current model is a student model. The student model is used with the model tracing algorithm to provide a diagnosis of the student, which is the trace of rules fired. Our system attaches some extra information in the form of hint and buggy messages to provide feedback from this trace. However, a better system would pass this trace to a cognitive model of a human tutor. This model would take the trace and decide what feedback to give the student. Using the second tutor model is more flexible, in that the tutor model can decide to give immediate or delayed feedback, or change the feedback based on other factors. For instance, the tutor model might decide to give no feedback, if it thinks that the student will discover the mistake on their own. In our domain, say a student moves into a room without clearing it first. There's a good chance they will be shot if there are opposing forces in the room. Model tracing the student model will find that

the moving into the room is incorrect, but the tutor model may decide not to say anything if knows there are opposing forces in the room. When the student is shot, they will hopefully learn their lesson. Adding a second tutor model would be fairly easy in that no changes would have be made to the existing student model or the model tracing algorithm, however the development of a tutor model would not be a simple task.

7 Conclusion

One goal of this thesis was to determine if model tracing is a feasible solution to tutoring a real-time collaborative task. By building a prototype architecture we found it was feasible, and found some evidence that our approach would scale to more complex models. Another goal of this thesis 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 thesis, it would not be hard to extend these some of these existing models into models that can also be used as tutors.

References

[1] B. S. Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13(6):4–16, 1984.

[2] P.A. Cohen, J.A. Kulik, and C.C. Kulik. Educational outcomes of tutoring: a meta-analysis of findings. *American Educational Research Journal*, 19:237–248, 1982.

[3] 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.

[4] Kenneth R. Koedinger, John R. Anderson, William H. Hadley, and Mary A. Mark. Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education*, 8:30–43, 1997.

[5] Sharon Dugdale. The design of computer-based mathematics instruction. In J. H. Larkin and R. W. Chabay, editors, *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*, pages 11–45. Erlbaum, Hillsdale, NJ, 1992.

[6] A. Mitrovic and S. Ohlsson. Evaluation of a constraint-based tutor for a database language. *International Journal of Artifical Intelligence in Education*, 10(3-4):238–256, 1999.

[7] S. Ohlsson. Constraint-based student modeling. In *Student Modeling: the Key to Individualized Knowledge-based Instruction*, pages 167–189. Springer, 1994.

[8] 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.

[9] 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.

[10] A. Corbett and J. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modeling and User-Adapted Interaction*, 4:253–378, 1995.

[11] Glenn Gunzelmann and Kevin A. Gluck. Knowledge tracing for complex training applications: Beyond bayesian mastery estimates.

[12] A. Newell. *Unified Theories of Cognition*. Harvard, Cambridge, MA, 1990.

[13] John R. Anderson. *Rules of the Mind*. Erlbaum, Hillsdale, NJ, 1993.

[14] W. Zachary and J. C. Le Mentec. A framework for developing intelligent agents based on human information processing architecture. In *1999 IASTED International Conference ASC*, pages 1–5, 1999.

[15] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[16] Peter Jackson. *Introduction to Expert Systems*. Addison Wesley Longman, Harlow, England, third edition, 1999.

[17] M. Singley, P. Fairweather, and S. Swerling. Team tutoring systems: Reifying roles in problem solving. In *Conference on Computer Supported Collaborative Learning*, 2000.

[18] Natalia López, Manuel Núñez, Ismael Rodríguez, and Fernando Rubio. Including malicious agents into a collaborative learning environment. In S. A. Cerri, G. Gouardères, and F. Paraguaçu, editors, *6th International Conference on Intelligent Tutoring Systems*, pages 51–60, 2002.

[19] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[20] Soldier visualization system. `http://www.ais-sim.com/svs.htm`.

[21] David Diller, William Ferguson, Alice Leung, Brett Benyo, and Dennis Foley. Behavior modeling in commerical games. In *Proceedings of the Thirteenth Conference on Behavior Representation in Modeling and Simulation*, pages 257–268, 2004.

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

[23] 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.

[24] 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.

[25] Robert A. Richards. Principle hierarchy based intelligent tutoring system for common cockpit helicopter training. In S. A. Cerri, G. Gouardères, and F. Paraguaçu, editors, *6th International Conference on Intelligent Tutoring Systems*, pages 473–483, 2002.

[26] Benjamin Bell, Joan Ryder, and Jonathan Cain. The good fly young: Augmenting undergraduate pilot training with selective-fidelity situation awareness training.

[27] W. Zachary, J. Cannon-Bowers, J. Burns, P. Bilazarian, and D. Krecker. An advanced embedded training system (aets) for tactical team training. In B.P. Goettl, H.M. Halff, C.L. Redfield, and V.J. Shut, editors, *Proceedings of the 4th International Conference on Intelligent Tutoring Systems*, pages 544–553. Berlin: Springer Verlag, 1998.

[28] Sanket Dinesh Choksey. Developing an affordable authoring tool for intelligent tutoring systems. Master's thesis, Worcester Polytechnic Institute, Worcester, MA, 2004.

[29] 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.

[30] 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.

[31] Neil T. Heffernan. *Intelligent Tutoring Systems have Forgotten the Tutor: Adding a Cognitive Model of Human Tutors.* PhD thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University, 2001. Technical Report CMU-CS-01-127.

[32] Michael van Lent, Ryan McAlinden, Paul Brobst, Barry G. Silverman, Kevin O'Brien, and Jason Cornwell. Enhancing the behaviorial fidelity of synthetic entities with human behavior models. In *Proceedings of the Thirteenth Conference on Behavior Representation in Modeling and Simulation*, pages 125–133, 2004.

[33] F. Ritter and W. Feurzeig. Teaching real-time tactical thinking. In J. Psotka, L. D. Massey, and S. A. Mutter, editors, *Intelligent Tutoring Systems: Lessons Learned*, pages 285–301. Erlbaum, Hillsdale, NJ, 1988.

A Cognitive Model

```
; Cognitive Model for Warrior Tutoring Project
; Copyright 2004 Tom Livak
; ========================================================================

; Working Memory Structure
; ========================================================================

(deftemplate setup
)

; -- Teams ---------------------------------------------------------
(deftemplate self
    (slot name)
    (slot goal)
    (slot room)
    (slot team)
)

(deftemplate person
    (slot name)
    (slot room)
    (slot type)
)

(deftemplate soldier extends person
    (slot team)
    (slot pos)
)

(deftemplate team extends setup
    (slot name)
    (slot type)
    (slot leader)
    (multislot team)
)

; -- Map -----------------------------------------------------------
(deftemplate room extends setup
    (slot name)
    (slot type)
    (slot status)
    (multislot doors)
)
```

```
(deftemplate door extends setup
    (slot name)
    (slot type)
    (slot room-from)
    (slot room)
    (multislot positions)
    (multislot paths)
)

(deftemplate path extends setup
    (slot name)
    (slot node)
    (slot pos (type INTEGER))
)

; -- Events ------------------------------------------------------------
(deftemplate event
)

(deftemplate at-event extends event
    (multislot nodes)
)

(deftemplate ready-event extends event
    (slot name)
)

(deftemplate clear-room-event extends event
    (slot door)
    (slot team)
)

(deftemplate clear-building-event extends event
    (slot team)
)

(deftemplate person-died-event extends event
    (slot name)
)

(deftemplate person-changed-room-event extends event
    (slot name)
    (slot room)
)

(deftemplate person-changed-type-event extends event
```

```
    (slot name)
    (slot type)
)

; -- Actions -------------------------------------------------------
(deftemplate action
    (slot message)
)

(deftemplate move-action extends action
    (slot node)
)

(deftemplate ready-action extends action
    (multislot team)
)

(deftemplate clear-room-action extends action
    (slot door)
    (slot team)
)

(deftemplate say-get-down-action extends action
)

(deftemplate say-move-out-action extends action
)

(deftemplate shoot-person-action extends action
    (slot person)
)

; -- Feedback ------------------------------------------------------
(deftemplate feedback
)

(deftemplate highlight-message extends feedback
    (slot node)
)

(deftemplate advice-message extends feedback
    (slot message)
)

; -- Goals ---------------------------------------------------------
(deftemplate goal
```

```
    (slot complete (default FALSE) )
    (multislot subgoals )
)

(deftemplate idle-goal extends goal
)

(deftemplate move-goal extends goal
    (slot position (type INTEGER) (default 0))
    (slot waiting (default FALSE))
    (slot path)
    (multislot team)
)

(deftemplate wait-goal extends goal
    (slot said-ready (default FALSE))
    (multislot waiting-on)
)

(deftemplate clear-room-goal extends goal
    (slot door)
    (slot step (default 1))
    (slot team)
)

(deftemplate control-civilians-goal extends goal
)

(deftemplate clear-building-goal extends goal
    (slot team)
    (multislot last-room)
)

; Rules
; ============================================================================

; clear-building
; ----------------------------------------------------------------------------

; if you are told to clear a building, and you're in charge of
; teamA, set a goal to clear the building

(defrule clear-buildingA0
    (self (name ?self-name))
    ?event <- (clear-building-event)
```

```
    (team (name team1A) (leader ?self-name))
=>
    (retract ?event)

    (assert (advice-message
        (message "You've been ordered to clear a building")
    ) )

    (bind ?new-goal (assert (clear-building-goal
        (team team1A)
        (last-room )
    ) ) )
)

; if you are told to clear a building, and you're in charge of
; teamB, set a goal to control civilians

(defrule clear-buildingB0
    (self (name ?self-name) )
    ?event <- (clear-building-event)

    (team (name team1B) (leader ?self-name))
=>
    (retract ?event)

    (assert (advice-message
        (message "You've been ordered to watch civilians")
    ) )

    (bind ?new-goal (assert (control-civilians-goal
    ) ) )
)

; if your goal is to clear a building, and there's an
; uncleared room, order your team to clear that room

(defrule clear-building1-decision "Commanding team"
    ?self <- (self (name ?self-name) (room ?room))
    (room (name ?room) (doors $? ?door $?))

    (door (name ?door) (room ?next-room))
    (room (name ?next-room) (status uncleared))

    ?goal <- (clear-building-goal (subgoals )
                (team ?team-name)
                (last-room $?room-stack)
```

```
    )

    (test (neq ?room (nth$ (length$ $?room-stack) $?room-stack)))

=>
    (assert (advice-message
        (message (str-cat "You need to order a team to clear " ?door))
    ) )

    (assert (clear-room-action
        (message "commanding team")
        (door ?door)
        (team ?team-name)
    ))

    (modify ?self (goal ?goal))
    (modify ?goal (last-room ?room $?room-stack))
)

; if your goal is to clear a building, and there's no
; more uncleared room, backtrack

(defrule clear-building2
    ?self <- (self (name ?self-name) (room ?room))

    (room (name ?room) (doors $? ?door $?))
    (door (name ?door) (room ?last-room))

    ?goal <- (clear-building-goal (subgoals )
                (team ?team-name)
                (last-room ?last-room $?room-stack)
    )

    (not (and
        (room (name ?room) (doors $? ?doorX $?) )
        (door (name ?doorX) (room ?next-roomX) )
        (room (name ?next-roomX) (status uncleared) )
    ) )
=>
    (assert (advice-message
        (message (str-cat "You need to backtrack" ))
    ) )

    (modify ?self (goal ?goal) (room ?last-room))
    (modify ?goal (last-room $?room-stack))
)
```

```
; if your goal is to clear a building, and there's an
; no uncleared rooms left, you are finished

(defrule clear-building3 "clear-building3"
    (self (name ?self-name) (room ?room))

    ?goal <- (clear-building-goal (subgoals ) (complete FALSE)
                (last-room )
    )

    (not (and
        (room (name ?room) (doors $? ?door $?) )
        (door (name ?door) (room ?next-room) )
        (room (name ?next-room) (status uncleared) )
    ) )
=>
    (assert (advice-message
        (message (str-cat "You finished clearing the building"))
    ) )

    (modify ?goal (complete TRUE) )
)

; control civilians
; ---------------------------------------------------------------------

; if your goal is to control civilians, and there's
; standing civilians in your room, order them to
; get down

(defrule control-civilians-decision "Securing Civilians"
    ?goal <- (control-civilians-goal)

    (self (name ?self-name))
    (soldier (name ?self-name) (room ?room-name))

    (person (room ?room-name) (type standing))
=>
    (assert (advice-message
        (message "You need control the civilans")
    ) )

    (assert (say-get-down-action
        (message "securing civilians")
    ) )
```

```
)

; clear-room
; ---------------------------------------------------------------------

; if ordered to clear a room, set a goal to clear that room

(defrule clear-room0
    ?event <- (clear-room-event (door ?door) (team ?team-name))
    (self (name ?self-name) (goal ?goal))
=>
    (retract ?event)

    (assert (advice-message
        (message "You've been ordered to clear a room")
    ) )

    (bind ?new-goal (assert (clear-room-goal
        (door ?door)
        (team ?team-name)
        (step 1)
    ) ) )

    (if (neq ?goal nil) then (modify ?goal (subgoals ?new-goal)))
)

; if your goal is clear a room, you first need to go to
; your stacking position outside the door, and wait
; for everyone else to stack

(defrule clear-room1-decision "Stacking"
    (self (name ?self-name))
    (soldier (name ?self-name) (pos ?man))
    (team (name ?team-name) (team $?team))

    (door (name ?door) (positions $? ?man ?path $?) )

    ?goal <- (clear-room-goal  (subgoals )
                (door ?door)
                (team ?team-name)
                (step 1)
    )
=>
    (assert (advice-message
        (message "You need to move into position")
    ) )
```

```
    (bind ?new-goal1 (assert (move-goal
        (path ?path)
        (team $?team)
    ) ) )

    (bind ?new-goal2 (assert (wait-goal
        (waiting-on $?team)
    ) ) )

    (modify ?goal (step 2) (subgoals ?new-goal1 ?new-goal2) )
)

; if your goal is clear a room, and everyone is stacked
; in the room, enter the room and move to your position
; inside

(defrule clear-room2-decision "Assaulting the room"
    (self (name ?self-name))
    (soldier (name ?self-name) (pos ?man))
    (team (name ?team-name) (team $?team))

    (door (name ?door) (paths $? ?man ?path $?) )

    ?goal <- (clear-room-goal (subgoals )
                (door ?door)
                (team ?team-name)
                (step 2)
    )
=>
    (assert (advice-message
        (message "You need to enter the room")
    ) )

    (bind ?new-goal1 (assert (move-goal
        (path ?path)
        (team $?team)
    ) ) )

    (bind ?new-goal2 (assert (wait-goal
        (waiting-on $?team)
    ) ) )

    (modify ?goal (step 3) (subgoals ?new-goal1 ?new-goal2) )
)
```

```
; if your goal is clear a room, you're the team leader,
; and everyone is in the room, and there are civilians
; in the room, you need to take command

(defrule clear-room3-decision "Taking command"
    (door (name ?door) (room ?room-name))
    ?room <- (room (name ?room-name))
    ?self <- (self (name ?self-name))

    ?goal <- (clear-room-goal (subgoals )
                (team ?team-name)
                (door ?door)
                (step 3)
                (complete FALSE)
    )

    (team (name ?team-name) (leader ?self-name))

    (not (person (room ?room-name) (type hostile)))
    (person (room ?room-name) (type standing))
=>
    (assert (advice-message
        (message "You need to take command")
    ) )

    (assert (say-move-out-action
        (message "taking command")
    ) )
)

; if your goal is clear a room, and the room is
; clear of enemies and civilians, you are done

(defrule clear-room4
    (door (name ?door) (room ?room-name))
    ?room <- (room (name ?room-name))
    ?self <- (self (name ?self-name))

    ?goal <- (clear-room-goal (subgoals )
                (team ?team-name)
                (door ?door)
                (step 3)
                (complete FALSE)
    )

    (not (person (room ?room-name) (type standing)))
```

```
    (not (person (room ?room-name) (type hostile)))
=>
    (assert (advice-message
        (message "You're done clearing the room")
    ) )

    (modify ?self (room ?room-name))
    (modify ?room (status cleared))

    (modify ?goal (complete TRUE) )
)


; move
; ---------------------------------------------------------------------

; if your goal is move somewhere, and you're ready to
; to move to the next on the path, than move

(defrule move-goal1-decision "Knowing where to move"
    (path (name ?name) (node ?node) (pos ?n))
    (self (name ?self-name))

    ?goal <- (move-goal (subgoals )
                (position ?n)
                (path ?name)
                (waiting FALSE)
    )
=>
    (assert (highlight-message
        (node (str-cat ?node))
    ) )

    (assert (move-action
        (message "moving into position")
        (node ?node)
    ))

    (modify ?goal (waiting TRUE) )
)

; if your goal is move somewhere, and you're arrived
; at the next node, than make note that you're ready
; to move to the next node.

(defrule move-goal2
```

```
    (path (name ?name) (node ?node) (pos ?n))
    ?event <- (at-event (nodes $? ?node $?))
    (self (name ?self-name))

    ?goal <- (move-goal (subgoals )
                (position ?n)
                (path ?name)
                (waiting TRUE)
    )
=>
    (retract ?event)

    (modify ?goal (position (+ ?n 1)) (waiting FALSE))
)

; if your goal is move somewhere, and you're arrived
; at the final destination, you're done

(defrule move-goal3
    (self (name ?self-name))

    ?goal <- (move-goal (subgoals )
                (position ?n)
                (path ?name)
                (waiting FALSE)
                (complete FALSE)
    )

    (not (path (name ?name) (pos ?n)))
=>
    (modify ?goal (complete TRUE))
)

; waiting
; ------------------------------------------------------------------------

; if you're waiting for your team, and someone say they
; are ready, make note that they are ready

(defrule person-ready
    ?goal <- (wait-goal (waiting-on $?n1 ?name $?n2) )
    ?event <- (ready-event (name ?name))
    (self (name ?self-name))
=>
    (retract ?event)
```

```
    (modify ?goal (waiting-on $?n1 $?n2) )
)

; if you're are doing something that requires waiting
; for everyone, and you're ready for to move on,
; tell your teammates

(defrule say-ready-decision "Proper communication"
    ?goal <- (goal (subgoals ?subgoal))
    ?subgoal <- (wait-goal (said-ready FALSE))

    (self (name ?self-name))
=>
    (assert (advice-message
        (message "Tell everyone you're in position")
    ) )

    (assert (ready-action
        (message "saying in position")
    ) )

    (modify ?subgoal (said-ready TRUE) )
)

; BUGGY RULE
; if you're are doing something that requires waiting
; for everyone, and you're NOT ready for to move on,
; tell your teammates

(defrule say-ready-buggy-decision "Improper communication"
    (buggy)

    ?goal <- (goal (subgoals $? ? ?subgoal $?))
    ?subgoal <- (wait-goal)

=>
    (assert (advice-message
        (message "BUG: You said in position when you were not!")
    ) )

    (assert (ready-action
    ) )
)

; if you're are doing something that requires waiting
; for everyone, and everyone is ready, you're done
```

```
; waiting

(defrule wait-done
    ?goal <- (wait-goal (waiting-on ) (said-ready TRUE) (complete FALSE) )

    (self (name ?self-name))
=>
    (modify ?goal (complete TRUE))
)




; persons
; ---------------------------------------------------------------------

; if there's an enemy in the room, shoot him

(defrule shoot-hostile-decision "Engaging enemy"
    (self (name ?self-name))
    (soldier (name ?self-name) (room ?room))

    (person (name ?person-name) (type hostile) (room ?room) )
=>
    (assert (advice-message
        (message "You need to engage the enemy")
    ) )

    (assert (shoot-person-action
        (message "engaging enemy")
        (person ?person-name)
    ) )
)

; BUGGY RULE
; if there's a civilian in the room, shoot him

(defrule shoot-other-buggy-decision "Violating ROE"
    (buggy)

    (self (name ?self-name))
    (soldier (name ?self-name) (room ?room))
    (person (name ?person-name) (type standing|down) (room ?room) )
=>
    (assert (advice-message
        (message "BUG: Do not shoot unarmed civilians!")
```

```
    ) )

    (assert (shoot-person-action
        (person ?person-name)
    ) )
)

; misc
; ------------------------------------------------------------------

; if a person died, update working memory

(defrule person-died
    ?event <- (person-died-event (name ?name))
    ?person <- (person (name ?name))

    (self (name ?self-name))
=>
    (retract ?event)
    (retract ?person)
)

; if a person changed rooms, update working memory

(defrule person-changed-room
    ?event <- (person-changed-room-event (name ?name) (room ?room))
    ?person <- (person (name ?name))

    (self (name ?self-name))
=>
    (retract ?event)
    (modify ?person (room ?room))
)

; if a person changed type, update working memory

(defrule person-changed-type
    ?event <- (person-changed-type-event (name ?name) (type ?type))
    ?person <- (person (name ?name))

    (self (name ?self-name))
=>
    (retract ?event)
    (modify ?person (type ?type))
)
```

```
; if you have a goal, and one it's subgoals is complete
; than make some mental notes

(defrule advance-goal
    ?goal <- (goal (subgoals $?s1 ?subgoal $?s2) )
    ?subgoal <- (goal (complete TRUE) )
    (self (name ?self-name))
=>
    (retract ?subgoal)
    (modify ?goal (subgoals $?s1 $?s2) )
)
```