Dynamically Altering Agent Behaviors Using Natural Language Instructions

Rama Bindiganavale, William Schuler, Jan M. Allbeck,
Norman I. Badler, Aravind K. Joshi, Martha Palmer
University of Pennsylvania
200 S. 33rd St.
Philadelphia, PA 19104
1-215-898-1976

{rama, schuler, allbeck, badler, joshi, mpalmer} @graphics.cis.upenn.edu

ABSTRACT

Smart avatars are virtual human representations controlled by real people. Given instructions interactively, smart avatars can act as autonomous or reactive agents. During a real-time simulation, a user should be able to dynamically refine his or her avatar's behavior in reaction to simulated stimuli without having to undertake a lengthy off-line programming session. In this paper, we introduce an architecture, which allows users to input immediate or persistent instructions using natural language and see the agents' resulting behavioral changes in the graphical output of the simulation.

Keywords

Autonomous agents, natural language processing, smart avatars, virtual environments.

1. INTRODUCTION

In this paper, we describe *smart avatars* [28, 34], which are virtual representations of humans controlled by real people. Given instructions interactively, smart avatars can act as autonomous or reactive agents. During a real-time simulation, a user should be able to dynamically refine his or her avatar's behavior in reaction to simulated stimuli without having to undertake a lengthy off-line programming session. Moreover, we would like to be able to instruct the smart avatars in a natural and straightforward manner.

For controlling, manipulating, and animating virtual humans, we need an effective user interface. Interactive point-and-click tools (such as Maya from Alias/Wavefront, 3D StudioMax from Autodesk, and SoftImage from Avid) could be configured to give instructions during run time, but require specialized training and animation skills, and force the user's instructions for the avatar through a narrow communication channel of hand/mouse motions. This narrow channel precludes, among other things, any conditional instruction that references a hypothetical object in its condition. For example, a user could not give the instruction "*if a military vehicle enters the checkpoint, open the gate*," if there is no such vehicle to click on when the instruction is given.

A programming or scripting language such as that used in Improv [25] can provide a sufficiently powerful interface for instructing virtual humans, and would be able to handle the kinds of conditional instructions discussed above. However, such interfaces are generally more appropriate for off-line applications, where instructions are written in advance, than for run-time applications, where instructions are given in response to observed events in the simulation. Not only does a scripting interface require a great deal of specialized programming expertise (even for "English-like" languages), but the required instructions can often be too complex to reliably implement in real time. A script to simply "*turn off all the lights*," for example, would involve at least one condition for testing whether each light was on, nested inside a loop over all the lights in the environment, in addition to the "*turn off light*" command itself.

One promising and relatively unexplored option for giving run-time instructions to virtual humans is a natural language based interface. After all, instructions for real humans are given in natural language, augmented with graphical diagrams and, occasionally, animations. Recipes, instruction manuals, and interpersonal conversations all use natural language as a medium for conveying information about processes and actions [3, 33]. A natural language interface should be powerful enough to express conditional instructions and hypothetical situations such as those described above, and should be simple enough to use in a real-time application without substantial formal training on the part of the user. We are not advocating that animators throw away their tools, only that natural language offers a communication medium we all know and can use to efficiently formulate run-time instructions for virtual human characters. Some aspects of some actions are certainly difficult to express in natural language (such as precise locations and orientations of objects), but the availability of a language interpreter can make the virtual human interface more closely simulate real interpersonal communication.

We have therefore implemented an architecture, which allows users to input instructions using natural language sentences. These instructions can range from specific instantaneous commands, like "*Sit down*," to very general standing orders, like

"*Drive abandoned vehicles to the parking lot*," affording various degrees of autonomy to the avatar/agent.

Existing natural language interface systems for agents either are not concerned with mapping natural language sentences into semantic representations for agent control [15] or rely on mostly ad hoc linguistic representations with severely limited syntactic processing [32] or no syntactic processing at all [9]. In contrast, the syntactic and semantic representations described in this paper are based as much on linguistic principles as they are on the needs of animation. Moreover, this system integrates detailed syntactic and semantic representations (such as those proposed in the AnimNL project [10]), with a broad-coverage grammar for English [35]. This grammar contains over 300,000 inflected English words with over 1,000 syntactic structures. Statistical techniques exist that find the correct word attachments 85% of the time, and the correct sentential analysis 35% of the time on unrestricted text using this grammar [35].

We begin by describing the levels of architectural control necessary for animating and instructing smart avatars with a rich set of behaviors in a dynamic environment. Then we discuss a conceptual representation and an architecture that we have created to facilitate instructing smart avatars with natural language. We close with an example simulation that was implemented under this architecture.

2. LEVELS OF ARCHITECTURAL CONTROL

Building a virtual human model that admits control from sources other than direct animator manipulations requires an architecture that supports higher-level expressions of movement. Although layered architectures for autonomous beings are not new [6, 16, 36], we have found that a particular set of architectural levels seems to provide efficient localization of control for both graphics and language requirements. Our multilevel architecture is grounded in typical graphical models and articulation structures [8, 12]. Motion generators drive these graphical models to generate various motor skills, and endow virtual humans with useful abilities. The higher architectural levels organize these skills with parallel automata, use a conceptual representation to describe the actions a virtual human can perform, and finally create links between natural language and action animation.

Our parallel programming model for virtual humans is called Parallel Transition Networks, or PaT-Nets [14]. Other human animation systems, including Motion Factory's Motivate and New York University's Improv [25], have adopted similar paradigms with alternative syntactic structures. In general, network nodes represent processes. Arcs connect the nodes and contain predicates, conditions, rules, and other functions that trigger transitions to other process nodes. Synchronization across processes or networks is made possible through message-passing or global variable blackboards to let one process know the state of another process.

PaT-Nets are effective programming tools, but do not represent exactly the way people conceptualize a particular situation. We therefore need a higher-level representation to capture additional information, parameters, and aspects of human action [13]. We create such representations by incorporating natural-language semantics into a Parameterized Action Representation.

3. PARAMETERIZED ACTION REPRESENTATION (PAR)

A PAR [2, 4] gives a description of an action. It specifies the action's agent, as well as any relevant objects and information about the path, location, manner, and purpose. There are linguistic constraints on how this information can be conveyed in a language: agents and objects tend to be verb arguments, paths are often prepositional phrases, and manners and purposes might be in additional clauses [24]. A parser and translator map the components of an instruction into the parameters or variables of the PAR, which is then linked directly to PaT-Nets executing the specified movement generators.

Natural languages often describe actions at a high level, leaving out many of the details that have to be specified for animation [23]. The PAR bridges the gap between natural language and animations.

We use the example "*Walk to the door and turn the handle slowly*" to illustrate the function of the PAR. There is nothing explicit in the linguistic representation about how to grasp the handle or which direction it will have to be turned, yet this information is necessary for the action's actual visible performance. The PAR has to include information about applicability and preparatory and termination conditions in order to fill in these gaps. It also has to be parameterized, because other details of the action depend on the agent, objects, and other attributes.

Next we briefly describe some of the terminology and concepts used to define a PAR and the architecture that we have designed to interpret it.

3.1. PAR Terminology

This is a sampling of the parameters in PAR:

- **Objects**: The object type is defined explicitly to represent a physical object and is stored hierarchically in a database (Section 3.2.3). Each object in the environment is an instance of this type and is associated with a graphical model in a scene graph.

  An object type lists the actions that can be performed on it and what state changes they cause [11, 20]. Among other fields, a list of grasp sites and directions are defined with respect to the object. These fields help orient actions that involve objects, such as grasping, reaching, and locomotion.

  In our example, "*Walk to the door and turn the handle slowly*," the "walk" action has an implicit floor as an object, while the "turn" action refers to the handle.

- **Agent**: The agent executes the action. The agents are treated as special objects, and their properties are stored in the hierarchical object database. Each agent is associated with an agent process (Section 3.2.5), which controls its actions based on the personality and capabilities of the agent. Not only does an agent's personality affect his or her response to a situation, but it also affects the way these

actions are performed. Two agents with different personalities would execute the same action in two different ways. For example, two agents could be waving at one another. A shy agent would wave his hand more slowly and with more hesitation than an extroverted agent would. This increases believability by preventing agents from reacting in the same manner in identical contexts and gives the impression that each agent has distinct emotions and personalities. The agent-specific parameters are specified through the graphical user interface and are resolved during the execution of the primitive action. In our example, the "walking" and "turning" actions share the same agent.

- **Applicability conditions**: The applicability conditions of an action specify what needs to be true in the world in order to carry out an action. These can refer to agent capabilities, object configurations, and other unchangeable or uncontrollable aspects of the environment. The conditions in this boolean expression must be true to perform the action. For "walk," one of the applicability conditions may be "Can the agent walk?" If these conditions are not satisfied, the action cannot be executed.
- **Preparatory Specifications**: This is a list of <*CONDITION, action*> statements. The conditions are evaluated first and have to be satisfied before the current action can proceed. If the conditions are not satisfied, then the corresponding action is executed—it may be a single action or a very complex combination of actions, but it has the same format as the execution steps described below. In general, actions can involve the full power of motion planning to determine, perhaps, that a handle has to be grasped before it can be turned. The instructions are essentially goal requests, and the smart avatar must then figure out how (if possible) it can achieve them. We currently specify the conditions to test for likely (but generalized) situations and execute appropriate intermediate actions. It would also be possible to add more general action planners, since the PAR represents goal states and supports a full graphical model of the current world state [31].

  In our example, one of the preconditions to be checked for the "walk" action is "stand" and the corresponding action is "stand up". If the agent is not standing, e.g., if he is sitting or prone, then the action causes him to change to the standing posture.
- **Execution Steps**: A PAR can describe either a primitive or a complex action. The execution steps contain the details of executing the action after all the conditions have been satisfied. If it is a primitive action, the underlying Pat-Net for the action is directly invoked. A complex action can list a number of sub-actions that may need to be executed in sequence, parallel, or a combination of both. A complex action can be considered done if all of its sub-actions are done or if its explicit termination conditions are satisfied.
- **Manner**: Manner specifications describe the way in which an agent carries out an action [1]. In our example, *slowly* modifies how the action "turn the handle" is performed.
- **Termination Conditions**: This is a list of conditions which when satisfied indicate the completion of the action. A termination condition can be determined from the main verb or attached clauses [5].
- **Post Assertions**: This is a list of statements or assertions that are executed after the termination conditions of the action have been satisfied. These assertions update the database to record the changes in the environment. The changes may be due to direct or side effects of the action.

A PAR takes on two different forms: uninstantiated (UPAR) and instantiated (IPAR). A UPAR contains *default* applicability conditions, preparatory specifications, and execution steps, but not information about the actual agent or physical objects involved. An IPAR is a UPAR instantiated with specific information on agent, physical object(s), manner, termination conditions, and other bound parameters. We store all instances of the UPARs in a hierarchical database called the *Actionary*™ (Section 3.2.3). Any new information in an IPAR overrides the corresponding UPAR default. An IPAR can be created from a natural language instruction (one or more IPAR for each new instruction) or dynamically by other PARs during execution.

3.2. PAR Architecture

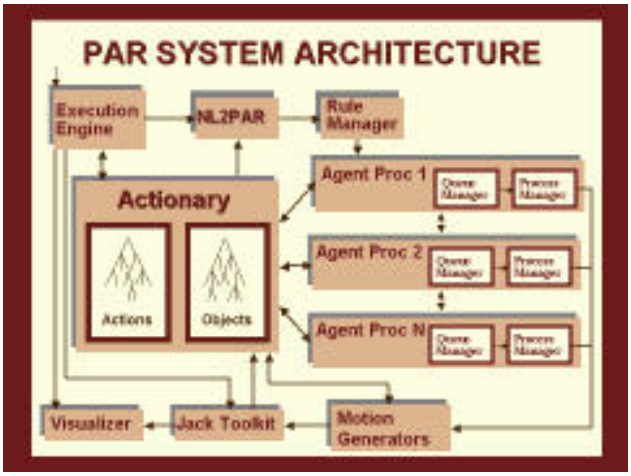

Figure 1: The PAR Architecture

3.2.1 Execution Engine

The execution engine is the main controller of the system. It maintains the global timer/controller, sends commands to the visualizer and Jack Toolkit to update the displayed scene, and sends user-input natural language instructions (inputted through a speech recognition system or through a graphical user interface (GUI)) to the NL2PAR module. The GUI also allows users to create and modify properties of the actions, objects, and agents.

3.2.2 NL2PAR

This module takes natural language instructions and then uses the Actionary to generate one or more instantiated PARs, which are then passed to the Rule Manager. The basic linguistic representation of an action is a *predicate-argument structure* such as '*slide(John, box),*' which indicates a particular action

(the predicate '*slide*') and its participants (the arguments '*John*' and '*box*').

We use the XTAG Synchronous Tree Adjoining Grammar system [27], which consists of a parser for extracting the predicate-argument structure of an input sentence (See also [24]), and a translator for generating an instruction script from this predicate-argument structure. The parser extracts these structures by first associating each word in an input sentence with one or more *elementary tree* fragments, which are combined into a single syntax tree for the entire input sentence using the constrained operations of the Tree Adjoining Grammar formalism [17, 18]. These elementary tree fragments have argument positions for the subjects and objects of verbs, adjectives, and other predicates, which constrain the way the fragments can be combined, and which determine the predicate-argument structure of the input sentence. The translator then converts this predicate-argument structure into an instruction script, which when executed generates one or more IPARs. With this architecture, a wide variety of inflections and grammatical transformations can be reduced to a much smaller set of predicates in the parser. This set of predicates can be further reduced to a still smaller set of PARs and scripting-language keywords in the translator. Although some parts of the translator may be domain-specific (some actions may depend on particular objects in a domain), the parser can easily be ported between domains, since its predicates are based on linguistic observations instead of on a particular programming language or virtual environment.

In order to support non-specific 'variable' references like "a military vehicle" or "abandoned vehicle" in instructions like, "*If a military vehicle enters the checkpoint, open the gate*," or "*Drive abandoned vehicles to the parking lot*," the translated scripts must incorporate nested 'for' statements that loop over every object in a set (in these cases, the set of vehicles). The purpose of the loops may be to find at least *one* object that satisfies a certain expression, or to ensure that *every* such object satisfies an expression, but in either case they must be bound successively to each object in a set. This means that the 'for' loop that binds such a variable must wrap around and therefore outscope the expression which contains that variable. Since we wish to generate scripts in languages that do not support lambda expressions, we prefer to use a transfer formalism that allows its compositional units to explicitly wrap around each other in the same way. For this purpose, we have adapted Synchronous Tree Adjoining Grammars [27, 29, 30], which makes use of the wrapping operation of 'adjunction' in Tree Adjoining Grammar to translate natural language instructions in English into executable scripts in *Python*. The implementation of 'variable' references described above is based on the TAG treatment of compositional semantics, and in particular quantifiers, developed in [19].

3.2.3 Actionary

All instances of physical objects, agents, and UPARs are stored in a pair of persistent hierarchical databases. One of the databases contains the objects and agents, and the other contains the UPARs. During the initialization phase of a simulation, a world model is created from the databases. This model is constantly updated during the simulation, recording any changes in the environment or in the properties of the agents and objects. The agent processes and motion generators can query the world model for the current state of the environment and for the current properties of agents and objects in the environment.

3.2.4 Rule Manager

The rule manager maintains a table of complex rules generated by the NL2PAR module. Users can give the system both immediate and persistent instructions (*standing orders*). For immediate instructions, the command scripts generated by the natural language interface are executed immediately. Depending on the results of test conditions and loops within a script, one or more instantiated PARs are generated and added to the appropriate agent's queue. In the case of standing orders, the scripts are stored as complex rules in a table. At each simulation frame, the rule manager evaluates each rule in the table and sends the resulting instantiated PARs, if any, to the appropriate agent process for execution. For example, the system might be executing the script for "*If a military vehicle enters the checkpoint, open the gate*," at each simulation frame, but only when a military vehicle actually enters the checkpoint would the script introduce an instantiated '*open gate*' PAR.

A persistent interpretation of "*Drive abandoned vehicles to the parking lot*," would work the same way, even though it does not contain an "*if*"-clause. At every frame, the script would iterate over all the vehicles in the checkpoint, but only when there actually is an abandoned vehicle would an instantiated '*drive*' PAR be introduced. It is important to note that each PAR added to an agent's action queue in this way would be instantiated with a specific vehicle in the checkpoint, and a specific destination point in the parking lot, even though none was specified in the natural language instruction. Each IPAR may also contain a number of preparatory actions, which move the agent to the relevant object (the vehicle to be driven). Taken together, the preparatory actions of all the introduced PARs describe a path between the different vehicles (if more than one), which was also not specified in the natural language instruction. In fact, although the agent is being instructed by a user, much of the agent's behavior is not specified in the instructions, but left for it to decide on its own. In this way, the high-level abstraction of a natural language interface gives the smart avatar a larger degree of autonomy in deciding how to carry out its instructions, which would be lost were its instructions completely specified through point-and-click or programming-language tools.

3.2.5 Agent Process

A separate agent process controls each instance of an agent. Each agent process has a queue manager that manages a priority-based multi-layered queue of all IPARs to be executed by the agent. The various tasks of an agent process are to:

- Add a given IPAR at the top level of the queue.
- Communicate with other agent processes through message passing.
- Trigger different actions for the agent based on the agent's personality, messages received from another agent process, and the existing environmental state.
- Return the process status (on queue, aborted/preempted, being executed, completed, etc.) of an IPAR.

- Ensure non-recursive addition of IPARs resulting from rules.

The queue manager in the agent process is implemented using PaT-Nets. Each UPAR is assigned a priority number by the user or by the situation. At any time, if the first action on the queue has a higher priority than the IPAR currently being executed, the queue manager preempts the current action. In general, either after preemption or completion of an action, a new action is selected to be popped from the top level of the IPAR queue of the agent and sent to a process manager. The selected new action has the highest priority in the first subset of monotonically increasing actions (with respect to priorities) at the beginning of the queue.

For each popped IPAR, a process manager first checks the termination conditions. If the termination conditions are already satisfied, then the action is not performed. If they are not satisfied, the applicability conditions are checked. If they are not satisfied, the entire process is aborted after taking care of failure conditions and proper system updates. If the applicability conditions are satisfied, the preparatory conditions are then checked. If any of the corresponding preparatory actions need to be executed, an IPAR is created (using the specified information of the UPAR, agent, and the list of objects) and added to the agent's existing queue of IPARs. It should be noted that the queue of IPARs is a multi-layered structure. Each new IPAR created for a preparatory action is added to a layer below the current one. The current action is continued only after the successful termination of all the preparatory actions. If the current action is very complex, more IPARs are generated and the depth of the queue structure increases. During the execution phase, a PaT-Net is dynamically created for each complex action specified in the execution steps or in the preparatory specifications. Each sub-action corresponds to a sub-net in the PaT-Net. The PaT-Nets are also used to ultimately ground the action in parameterized motor commands to the embodied character.

3.2.6 Motion Generators

For the execution of the primitive actions, the process manager invokes the corresponding pre-registered motion generators. It is within these motion generators that all the parameters of the IPAR are finally resolved.

These generators access the Actionary for information on agents, objects and the current state of the environment. During the execution of the action, the motion generators update the Actionary with the status of the ongoing action. At the end of the action, they use post assertions to update the Actionary with the various changes in the environment.

3.2.7 Toolkit and Visualizer

We use the *Jack*® toolkit and OpenGL® to maintain and control the actual geometry, scene graphs, and human behaviors and constraints.

3.3. PAR Implementation

We have implemented PAR using C++ and *Python* [22]. *Python* is an interpreted object-oriented language and is compatible with C++. This makes it easy for both the action and object hierarchies to be visible from both *Python* and C++. It also provides serialization and persistence that is ideal for database implementation.

The system is built in two layers. The bottom layer contains the implementation of the core system and is in C++ and *Python*. The top layer has a graphical user interface (GUI) built using Python-Tk. The user needs to interact only with the top layer. This allows for the UPARs and objects to be dynamically created. The applicability conditions, preparatory specifications, and executable actions are presently written through the GUI as simple Python scripts which can be easily tested. As *Python* can be extended and embedded in C++, objects of different data types can be passed between them. The objects passed from *Python* to C++ are all perceived by C++ to be of a single *Python* object type that could later be typecast to different types. This allows conditions to return the various test results as either a Boolean type or a Python string. The agent process is capable of expanding this string into a new action and adding it to the agent's queue.

4. EXAMPLE: PAR FOR VIRTUAL ENVIRONMENT TRAINING

PAR for Virtual Environment Training has been designed to demonstrate language-based control in the PAR system using the XTAG Synchronous TAG parser to generate PARs from complex natural language instructions.

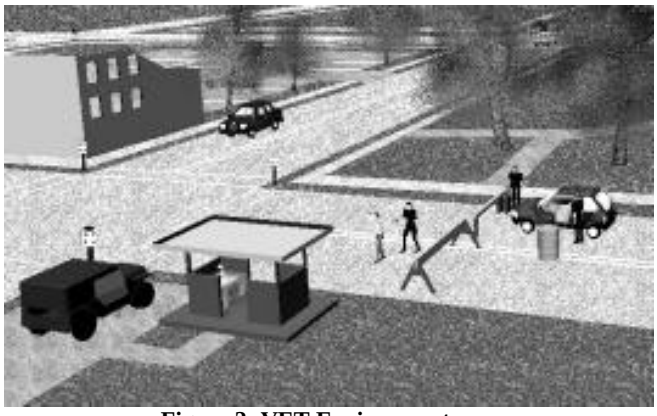

Figure 2: VET Environment

The simulation allows a user to give specific immediate instructions or standing orders to the virtual agents within the environment. This test application was designed to train soldiers for a military peace-keeping operation. Creating a virtual training environment with language-based control has many benefits over other possible training environments. First, it is less expensive than setting up a real, physical environment. In addition to this cost, there is also a cost associated with hiring and training actors to participate in the training secessions. Second, it allows for more variability than prerecorded video training. With prerecorded video, there is a fixed number of scenarios possible, and there are limitations on the viewpoint from which the trainees can see the situation. Third, our virtual training environment allows group training to be performed even when part of group is not available. Autonomous agents can represent any missing members of the group. Fourth, computer networks and distributed simulations provide the

ability to train with people in separate locations, perhaps even on a ship. Finally, the language-based control allows the soldiers to verbally interact with the system in a natural way.

The scenario (Figure 2) is centered on a military checkpoint at the edge of a town. There are three soldiers in training (henceforth referred to as trainees) whose job is to apprehend suspected terrorists. A separate agent process controls each of the trainees. A process simulator (an autonomous agent process) generates new vehicles, controls their movements, and operates traffic lights. As each vehicle approaches the checkpoint, one of the trainees checks each civilian driver's identification. If there is a match, the trainee is supposed to draw his weapon and take the driver into custody. All others are allowed to pass through the checkpoint.

During this process, the trainees may (inadvertently) commit different errors, which may result in one of the trainees getting shot by a driver. In an effort to correct the situation, the user can give different standing orders to the trainees.

4.1 Agents

The agents in this scenario are the three trainees, the drivers, and the process simulator. The PAR system eases the creation of autonomous virtual human agents by providing a flexible way of representing personalities, emotions, and other agent properties, and also by providing fast, straightforward ways to query the environment and communicate with other agents.

In this simulation, the virtual human agents have one of two personalities; hostile or non-hostile. Currently, the trainees are all non-hostile. Drivers are randomly chosen to be hostile or non-hostile as they enter the scene and their hostility can be changed at any time during the simulation. The personality of the driver is expressed through his actions. For example, a hostile driver is far more likely to be uncooperative and may even attempt to shoot one of the trainees.

Figure 3: Angry Facial Expression

In PAR, one of the properties of an agent is the emotional state of the agent. The agents are capable of being angry, sad, fearful, happy, or emotionally neutral. Initially, all of the trainees are emotionally neutral, and the drivers' emotions are set randomly as they enter the checkpoint. During the simulation, environmental conditions and the agent's actions effect the agent's emotions. For example, a trainee becomes angry whenever he draws his weapon. The emotions of the agents are expressed through facial expressions (Figure 3). We are currently using a facial animation model from MIRALab, University of Geneva [21].

In PAR, the agents externally communicate with each other using speech and gestures. Internally, their processes communicate through message passing. For example, when the driver hands his identification to the trainee, the trainee needs to respond by taking the identification. He knows to do this because the driver process sends the trainee process a message informing him that the identification is being handed over. This approach sidesteps questions of gesture recognition that may be addressed in the future.

4.2 Standing Orders

Standing orders are persistent natural language instructions issued to correct the trainees' errors.

In the first scenario, the trainee draws his weapon at a suspected terrorist, but forgets to take cover. The driver shoots him. To correct this, the user gives the following standing order to that trainee:

"*When you draw your weapon at the driver, take cover from the driver behind your drum.*"

The NL2PAR module immediately parses this standing order and the generated *Python* script is stored as a complex rule in the rule table. In the next trial of the simulation, as soon as the trainee draws his weapon the standing order (now a rule) forces him to also take cover behind his drum correcting the situation. Also, it is noticed that when one of the trainees draws his weapon, the other two trainees remain still, which is situationally incorrect. So, a standing order of

"*If Trainee1 draws his weapon at the driver, draw your weapon at the driver and take cover from the driver behind your drum*"

remedies this situation.

The command "*take cover*" takes two oblique arguments—the potential threat (in this case, the driver), and the desired cover (the steel drum). When this PAR is executed, the trainee moves to a place where the drum intersects the path between himself and the driver. But, since the "*take cover*" action is parameterized at this high level, the trainee could be instructed to take cover from virtually any object (say, from one of his companions), behind any other object (say, behind the suspect's car), and the simulation would accommodate it.

During another run of the system a hostile driver draws a gun as soon as the first trainee asks him for his identification, and shoots the trainee before he can react. Observing that the trainees on the passenger side of the car could have seen the driver reach for the gun, the user gives two additional standing orders for trainees 2 and 3:

"When there is a driver, watch the driver."

"If the driver reaches for a gun, warn Trainee1."

Once again the simulation is replayed and whenever there is a driver in the car at the checkpoint, the new standing orders force trainees 2 and 3 to watch the driver. When the driver reaches for his gun, the rules force those trainees to warn the first trainee, before the driver can grab his weapon and fire. This gives the first trainee time to draw his own weapon and take cover. Since all the previous orders are still in the system's memory, trainees 2 and 3 also draw their weapons, and all three take cover. The

driver, outnumbered, quickly surrenders, and the trainees successfully complete the exercise.

It is important to note that the '*watch'* action is classified as a preemptive action and so its UPAR has a lower priority. This means that whenever the trainees need to execute other actions, their agent processes will preempt '*watch'* from their queues and execute the other actions. But, after the other actions have been completed, if there is still a driver at the checkpoint, the rule resulting from the standing order "*When there is a driver, watch the driver"* will again force the trainees to watch the driver. This results in a completely natural looking scenario where the trainees are always cautiously watching the driver. If they are interrupted to do something else, they quickly finish that task and resume watching.

5. DISCUSSION

The PAR architecture and its implementation is intended to provide a test bed for real-time agents who work, communicate, and manipulate objects in a synthetic 3-D world. Our goal is to make interaction with these embodied characters the same as with live individuals. We have focused on language as the medium for communicating instructions and finite state machines as the controllers for agent or object movements.

The structure described here is the basis for a new kind of dictionary we call an *Actionary*™. A dictionary uses words to define words. Sometimes it grounds concepts in pictures and (in on-line sources) maybe even sounds and video clips. But these are canned and not *parameterized* — flexible and adaptable to new situations the way that words function in actual usage. In contrast, the Actionary™ uses PAR and its consequent animations to ground action terms. It may be viewed as a 3-D (spatialized) environment for animating situated actions expressed in linguistic terms. The actions are animated to show the meaning in context, that is, relative to a given 3-D environment and individual agents.

Any simulation system must be supplied with procedures that implement its semantics. PAR is no exception. We have discussed how the multi-level approach lets us focus the system implementation in separable and re-usable motion generators, PaT-Nets, and PARs, but these must still be manually coded. People build their internal representations through experience and learning, but these avenues are not yet open to the simulation designer. We are developing tools that may allow UPARs and some of their parameters to be learned from observation. Additionally, we may be able to use the natural language interface to construct portions of a UPAR, just as a word dictionary provides some but not all of a term's semantics.

Currently, we are working on adding a planner [31] to the PAR system. Our hope is that the information stored in each UPAR, including the applicability conditions, preparatory specifications, termination conditions, and post assertions, will ease the task of planning and provide us with a robust, real-time simulation system.

The actions requested may fail for any number of reasons from failed conditions in the PAR to unanticipated confounding events to inadequate implementation. Since our agent models admit the possibility of adding action planners, some failures may be detectable and repairs initiated automatically [7,26]. Giving an agent the capability of observing the consequences of its actions (or inaction) and generating compensating rules is an attractive future effort.

An instruction understanding system, based on natural language inputs, an Actionary™ translation, and an embodied virtual human agent could provide a non-programming interface between real and virtual people. We can describe tasks for others and see them carried out, whether they are real or virtual participants. Thus the door is opening to novel applications for embodied agents in games and interactive entertainment, job training, team coordination, manufacturing and maintenance, education, and emergency drills. As the Actionary™ grows, new applications should become ever easier to generate. And just as our human experience lets real people take on new tasks, so too should embodied characters be adaptable to new environments, new behaviors, and new instructions.

6. ACKNOWLEDGEMENTS

We would like to acknowledge Liwei Zhao, Hogeun Shin, Seung-Joo Lee, Sooha Park Lee, Meeran Byun, Harold Sun, and Aaron Bloomfield for their work on this research. This research is partially supported by U.S. Air Force F41624-97-D-5002, Office of Naval Research K-5-55043/3916-1552793, DURIP N0001497-1-0396, and AASERTs N00014-97-1-0603 and N0014-97-1-0605, DARPA SB-MDA-97-2951001, NSF IRI95-04372, SBR-8900230, and IIS-9900297, Army Research Office ASSERT DAA 655-981-0147, NASA NRA NAG 5-3990, and Engineering Animation Inc.

7. REFERENCES

[1] Badler, N., D. Chi, and S. Chopra. 1999. Virtual human animation based on movement observation and cognitive behavior models. In *Proceedings of the Computer Animation 1998*, 128-137. Los Alamitos, CA: IEEE Computer Society.

[2] Badler, N., M. Palmer, and R. Bindiganavale. 1999. Animation control for real-time virtual humans. *Communications of the ACM* 42(7): 65–73.

[3] Badler, N., B. Webber, J. Kalita, and J. Esakov. 1990. Animation from instructions. In N. Badler, B. Barsky, and D. Zeltzer, eds., *Making Them Move: Mechanics, Control, and Animation of Articulated Figures*, 51–93. San Francisco, CA: Morgan-Kaufmann.

[4] Badler, N., B. Webber, M. Palmer, T. Noma, M. Stone, J. Rosenzweig, S. Chopra, K. Stanley, J. Bourne, and B. Di Eugenio. 1997. Final report to Air Force HRGA regarding feasibility of natural language text generation from task networks for use in automatic generation of Technical Orders from DEPTH simulations. Technical Report, CIS, University of Pennsylvania.

[5] Bourne, J., 1998. Generating adequate instructions: Knowing when to stop. In *Proceedings of the AAAI/IAAI Conference*, 1169. Doctoral Consortium Section, Madison, Wisconsin.

[6] Brooks, R., A robot that walks: Emergent behaviors from a carefully evolved network. *Neural Computation* 1(2): 253–262.

[7] Cassell, J., and H. Vilhjálmsson. 1999. Fully Embodied Conversational Avatars: Making Communicative Behaviors Autonomous. *Autonomous Agents and Multi-Agent Systems*, 2(1): 45-64.

[8] Cavazza, M., R. Earnshaw, N. Magnenat-Thalmann, and D. Thalmann. 1998. Motion control of virtual humans. *IEEE Computer Graphics and Applications* 18(5): 24–31.

[9] Chapman, D. 1990. *Vision, instruction, and action*. Ph.D. thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

[10] DiEugenio, B. and B. Webber. 1992. Plan recognition in understanding instructions. *In Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, 52—61.

[11] Douville, B., L. Levison, and N. Badler. 1996. Task level object grasping for simulated agents. *Presence* 5(4): 416–430.

[12] Earnshaw, R., N. Magnenat-Thalmann, D. Terzopoulos, and D. Thalmann. 1998. Computer animation for virtual humans. *IEEE Computer Graphics and Applications* 18(5): 20–23.

[13] Firby, R. 1994. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems*, 49—54, Chicago IL.

[14] Granieri, J., Becket, W., and Reich, B.D. 1995. Behavioral control for real-time simulated human agents. *Symposium on Interactive 3D Graphics*, 173—180.

[15] Huffman, S.B., and J.E. Laird. 1995. Flexibly instructable agents. *Journal of Artificial Intelligence Research*, 3: 271—324.

[16] Johnson, W.L. and J. Rickel. 1997. Steve: An animated pedagogical agent for procedural training in virtual environments. *SIGART Bulletin*, 8(1-4): 16—21.

[17] Joshi, A.K., 1985. How much context sensitivity is necessary for characterizing structural descriptions: Tree adjoining grammars. In D. Dowty, L. Karttunen, and A. Zwicky, eds., *Natural Language Parsing: Psychological, Computational and Theoretical Perspectives*, 206–250. Cambridge: Cambridge University Press.

[18] Joshi, A.K., and Y. Schabes. 1996. Tree-Adjoining Grammars and Lexicalized Grammars. *Handbook of Formal Languages and Automata*. Springer Verlag, Berlin, eds: A. Salomaa & G. Rosenberg, 409-431.

[19] Kallmeyer, L. and A. Joshi. 1999. Underspecified Semantics with LTAG. In *Proceedings of Amsterdam Colloquium on Semantics*.

[20] Kallmann, M., and D. Thalmann. 1999. A behavioral interface to simulate agent-object interactions in real-time. In *Proceedings of Computer Animation*, 138–146. Los Alamitos, CA: IEEE Computer Society.

[21] Kalra, P., A. Mangili, N. Magnenat-Thalmann, and D. Thalmann. 1992. Simulation of muscle actions using rational free form deformations. *Proceedings Eurographics '92, Computer Graphics Forum* 2(3): 59-69.

[22] Lutz, M. 1996. *Programming Python*. Sebastapol: O'Reilly.

[23] Narayanan, S. 1997. Talking the talk is like walking the walk. In *Proceedings of the 19th Annual Conference of the Cognitive Science Society*, 548-553. Palo Alto, Calif. Mahwah, NJ: Lawrence Erlbaum and Associates.

[24] Palmer, M., J. Rosenzweig, and W. Schuler. 1998. Capturing motion verb generalizations with synchronous tag. In P. St. Dizier, ed., *Predicative Forms in NLP*. Text, Speech, and Language Technology Series, 250-277. Dordrecht, The Netherlands: Muwer Press.

[25] Perlin, K., and A. Goldberg. 1996. Improv: A system for scripting interactive actors in virtual worlds. In *Computer Graphics*, 205-216. New York, NY: ACM SIGGRAPH.

[26] Rickel J., and W.L. Johnson 1999. Animated agents for procedural training in virtual reality: Perception, cognition and motor control. *Applied Artificial Intelligence* (13):343-382.

[27] Schuler, W. 1999. Preserving semantic dependencies in synchronous tree adjoining grammar. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, 88—95.

[28] Shi, J., T. J. Smith, J. Granieri, and N. Badler. 1999. Smart avatars in JackMOO. In *Proceedings of IEEE Virtual Reality*, 156–163. Los Alamitos, CA: IEEE Computer Society.

[29] Shieber, S. 1994. Restricting the weak-generative capability of synchronous tree adjoining grammars. *Computational Intelligence* 10(4).

[30] Shieber, S. and Y. Schabes. 1990. Synchronous tree adjoining grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*. Helsinki, Finland.

[31] Trias, T., S. Chopra, B. Reich, M. Moore, N. Badler, B. Webber, and C. Geib. 1996. Decision networks for integrating the behaviors of virtual agents and avatars. In *Proceedings of* IEEE. *Virtual Reality Annual International Symposium*. Los Alamitos, CA: IEEE Computer Society.

[32] Vere, S. and T. Bickmore. 1990. A basic agent. *Computational Intelligence*, 6: 1—22.

[33] Webber, B., N. Badler, B. Di Eugenio, C. Geib, L. Levison, and M. Moore. 1995. Instructions, intentions and expectations. *Artificial Intelligence Journal* 73:253–269.

[34] Wilcox, S, K. *Web Developer's Guide to 3D Avatars*. Wiley, New York, 1998.

[35] XTAG Research Group. 1998. A lexicalized tree adjoining grammar for English. Technical Report, University of Pennsylvania.

[36] Zeltzer, D. 1990. Task-level graphical simulation: Abstraction, representation, and control. In N. Badler, B. Barsky, and D. Zeltzer, eds., *Making Them Move: Mechanics, Control, and Animation of Articulated Figures*, 3–33. San Francisco, CA: Morgan-Kaufmann.