Proceedings, The Thirteenth AAAI Conference on
Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)

Dante Agent Architecture for Force-On-Force Wargame Simulation and Training

Brian Hart, Derek Hart, Russell Gayle, Fred Oppel, Patrick Xavier, Jonathan Whetzel
Sandia National Laboratories
PO Box 5800 Albuquerque, New Mexico, USA 87185-1004
{bhart, derhart, rgayle, fjoppel, pgxavie, jhwhetz}@sandia.gov

Abstract

Physical site security heavily relies on expert teams continually examining and testing security profiles for discovering potential vulnerabilities. These experts hypothesize scenario(s) of interest and conduct "red versus blue" simulated exercises where they execute tactics that might reveal possible dangers. Due to the intensive manpower required, video-game environments have become a widely-adopted mechanism for conducting these exercises with virtual agents replacing many of the human roles for quicker analyses. However, these agents either have limited capabilities or require several engineers to develop realistic behaviors. This paper documents an agent architecture and authoring suite that enables subject matter experts to easily build complex attack/response plans for agents to use within *Dante*, a 3D simulation platform for video-game-based training/analysis of force-on-force engagements. This work expands upon current trends in commercial video-game artificial intelligence (AI) architectures to build agent behaviors deemed qualitatively valid by security experts, with the runtime of these algorithms best suited for turn-based, strategy games.

Introduction

Wargaming, games of strategy that focus on military operations, remains as an effective tool for hypothesizing and testing various conflict scenarios, while also enjoying popularity in the commercial game market. Modern wargames have evolved by moving from physical tabletop environments to video game platforms, improving their manageability.

The complexity of these modern wargames can require several hours of training to gain familiarity, causing a shortage of well-trained operators to perform the exercise. Outcomes from the exercise depend on how well each of the operators can play the game, casting doubt on the results if the sides have a mismatch in operator experience with the game. To solve this issue, wargame developers have utilized AI agents to replace human players. These agents have been plagued with problems of predictability or inability to effectively handle novel situations, leading to high financial costs for engineers who must construct new agent behaviors.

Sandia National Laboratories has engaged in research toward developing an agent architecture for simulation and training within the domain of physical site security wargames; a strategy game where an opposition team attempts to reach an objective within a secured area manned by a defensive team. This research explores conflicting goals of: 1) developing agents with qualitatively-valid behaviors as judged by subject matter experts, and 2) ease of constructing agent behaviors by non-technical personnel. This paper describes the agent architecture devised for *Dante*, a force-on-force 3D simulation platform that can operate in closed-loop simulations (agent v. agent) and human-in-the-loop wargaming configurations (human v. agent, or human/agent hybrid teams). This research expands upon popular agent architecture designs within the entertainment sector for building more realistic behaviors without compromising on ease of development.

Umbra Simulation Framework

Dante is built on top of the modular *Umbra* Simulation Framework (Gottlieb, Harrigan, McDonald, Oppel, & Xavier, 2001), developed at Sandia National Laboratories for modeling, analysis and deployment of robots and intelligent system technologies for manufacturing, military, and security systems. Since its initial development in 1997, *Umbra* has proven its utility in complex adaptive systems engineering. In addition to representing a variety of intelligent systems including robots, unmanned air and ground vehicles, communications systems, and sensors, *Umbra* is being used to address other issues of human behavior including nuclear physical security, other hazardous environments, insider attacks, and active shooter events.

Umbra's main strength is it allows engineers and analysts to break complex system software problems into collections of manageable pieces or modules that can be modeled independently and then efficiently combined into a system. *Umbra* provides a software framework with a structured module update, a variety of base classes, and an interactive script-level interface that facilitates efficient and effective code development, debugging, reuse, experimentation, and deployment all within a 3D environment. In addition, *Umbra* includes an extensible core set of 3D graphical libraries to enable a user to efficiently create, view, and interactively interrogate entities from any viewing perspective within a 3D environment.

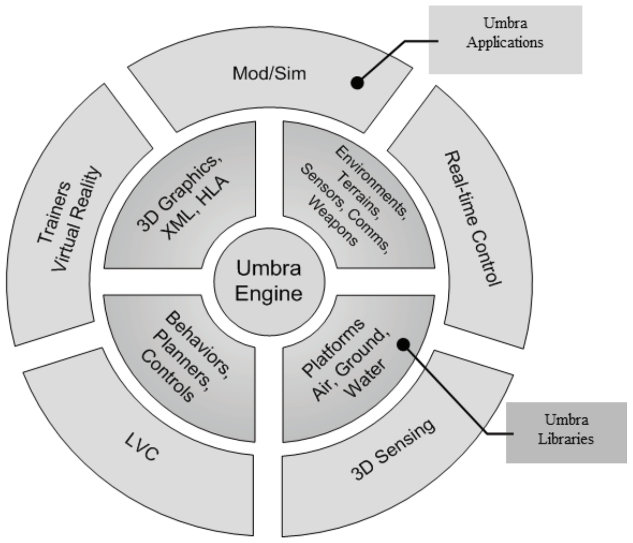

Figure 1: Umbra Simulation Framework & Applications

Related Work

Many research efforts in game AI have sought a balance between providing complex, realistic behaviors while providing ease of authoring. Traditional architectures such as finite state machines (FSMs) and behavior trees provide intuitive structures for designing agents, yet have difficulties in scaling the agents to handle more tasks and are too predictable in their behavior execution. Cognitive modeling solutions such as SOAR (Laird, 2012), ACT-R (Anderson & Lebiere, 1998), and Sandia's Cognitive Runtime Engine with Active Memory (SCREAM) (Djordjevich, et al., 2008), provided new architectures mimicking psychological models of human-decision making, with demonstrations of these models used as agents within various games (Laird, 2001) (Best, Lebiere, & Scarpinatto, 2002). These approaches have not met wide acceptance due to their complicated structure for authoring new behaviors. *Dante's* agent architecture has been inspired by our past efforts in cognitive modeling by improving the extensibility of an agent across novel environments.

Other commercial game AI structures have also been developed to make agent's more robust in both their behaviors and ease of authoring. Goal-Oriented Action Planning (Orkin, 2004) enables an agent to both determine its next goal and how to achieve it through planning a sequence of actions to execute. Other solutions have employed utility theory (Dill & Mark, 2010), selecting an action to perform by calculating every action's expected benefit value at any moment. These solutions improve upon traditional methods by allowing more flexibility in the agent's behavior selection with little effort needed to expand their capabilities. Yet, these architectures make it difficult for designers to manipulate agent behavior since the agents dynamically plan action sequences throughout their execution.

Machine learning-based approaches have seen considerable success recently across a wide spectrum of game types. With respect to tactical wargaming, successes have revolved around optimizing localized components (e.g., optimizing utility functions for a character in a skirmish) or for limited numbers of agents (McPartland & Gallagher, 2011). It remains to be seen how these approaches would scale to include more complex domains. Furthermore, with machine learning approaches, it is difficult to design or edit any generated behaviors (e.g. to be more human-like).

Dante's agent architecture handles these conflicting desires between realistic behaviors and ease of authoring by reducing the level of detail for an agent's planning. Decision-making for *Dante* agents resembles a utility theory approach, with the individual actions being replaced by FSMs that handle common behavior sequences. Other researchers have proposed search algorithms for behavior selection in real-time strategy games with success, such as the alpha-beta search for scripted behaviors (Churchill, Saffidine, & Buro, 2012) and goal-driven autonomy (Munoz-Avila, Aha, Jaidee, Klenk, & Molineaux, 2010). This paper builds upon these efforts by documenting the *Dante* agent architecture and how our users can construct agent teams for developing complex wargaming scenarios within our 3D simulation engine.

Dante Behavior Architecture

Characters in *Dante* simulations attempt to execute commands issued to them by human users or by a virtual Commander system that dynamically issues commands to characters during runtime. The execution of these commands by the character is done via a set of behaviors. Characters can participate in the execution of commands with other characters with coordination occurring between them. Commands have conditions that indicate their completion in either a success or failure state, which can trigger subsequent commands that formulate a long-term plan for the characters.

Behaviors in *Dante* are packages of actions (FSMs) that accomplish a specific goal. For example, the `Move` behavior contains the ability to plan a path to a goal location, the movement to that goal location via the path, and an orient at the end to ensure the character is facing the desired direction. Another example would be the `Engage` behavior, which includes the steps necessary to engage a threat using a weapon. In `Engage`, the character ensures the best weapon is equipped, aims that weapon, and fires that weapon. If the behavior continues, they can choose to reload the weapon as necessary and reevaluate the situation.

When a character is issued a command, they execute a priority queue of behaviors. The queue is sorted on the priority and activation of the behaviors available, calculated through a utility-theory AI approach. Behaviors can run concurrently or preempt other behaviors to take control of the character. Later, preempted behaviors can be restarted to accomplish the specified command.

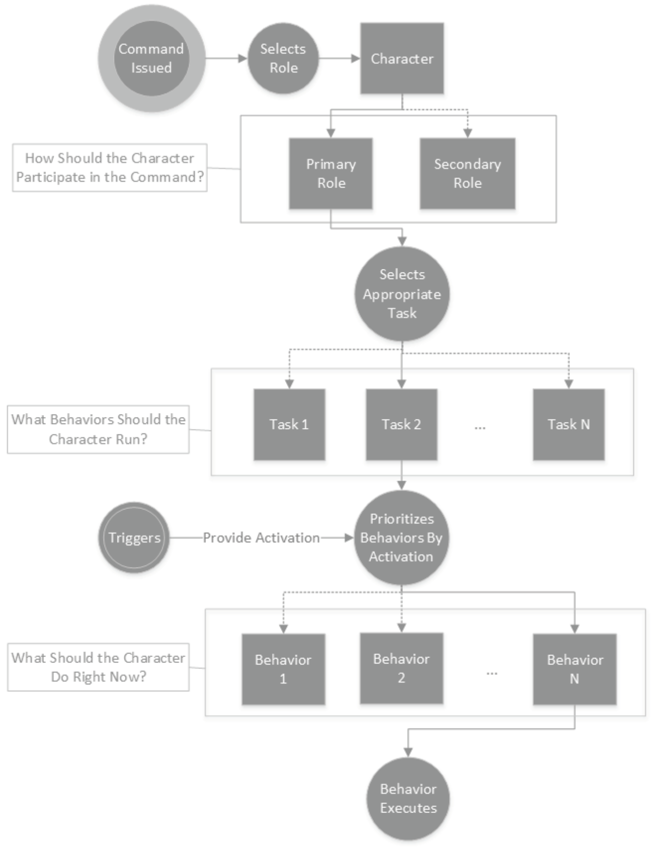

Figure 2: Diagram of the top-down Dante Behavior Engine

Commands, Roles, and Tasks

A Command accepts a set of characters to accomplish a specific goal. A Command can be issued to a single character or multiple characters. A Command is responsible for configuring the team of characters to suit its own needs. For some commands that configuration is trivial and requires no work, but others must divide the team into specific jobs to accomplish the command's goal. Commands also determine what success means and who must accomplish the goal before the command is complete.

Commands are notified when the characters working on them either succeed or fail. This gives the command a chance to deal with each character's completion. In the case of a character reporting success, it could mean that the job is now done and the plan can move forward. In the case of a character reporting failure, it means the command might need to adjust the team to compensate for the character who just failed (e.g., character has been killed and no longer available). Commands also have the option of completing even when all characters executing the command have not finished.

Roles determine how a character translates a Command into a set of behaviors to execute. Roles allow *Dante* characters to differentiate how they react to the same Command. When a Role receives a command, it translates the Command into a Task, with a Role having only one Task associated with it. A Task represents a collection of behaviors (FSMs) that can be executed, with the underlying Behavior Selector determining which behavior(s) should execute.

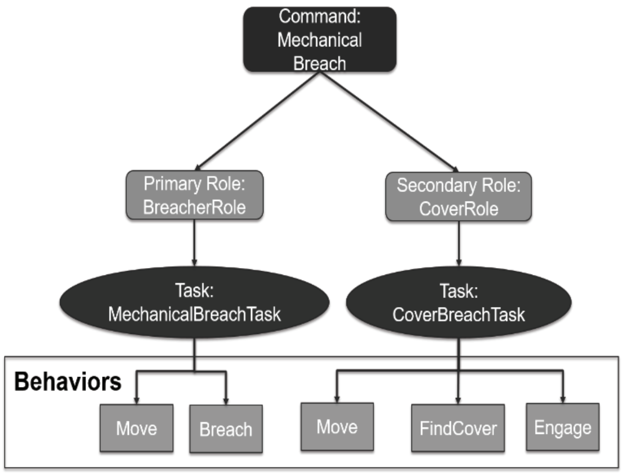
Command: Mechanical Breach
Primary Role: BreacherRole
Secondary Role: CoverRole
Task: MechanicalBreachTask
Task: CoverBreachTask
Behaviors
Move
Breach
Move
FindCover
Engage

Figure 3: Example of MechanicalBreach command issued to team

Figure 3 provides an example of a team receiving the command `MechanicalBreach`, where the goal is to make a hole in a physical barrier via a mechanical device (crowbar, wire cutters, etc.). `MechanicalBreach` consists of two distinct jobs. The first job is to do the work of the breach, but this requires only one character to perform the work. The second job is to provide cover for the breacher by engaging threats. The `MechanicalBreach` puts the character chosen to breach into the `BreacherRole` and everyone else into the `CoverRole`. The `BreacherRole` signifies the character should do the `MechanicalBreachTask` which includes the `Breach` behavior. The other characters using the `CoverRole` execute the CoverBreachTask, which includes the Engage behavior. That means we get one character doing the breach work, while the others are free to engage threats as necessary. Characters providing cover to the breacher may react to potential threats by finding a defensive position to safely return fire.

Behavior Selectors

Behavior Selectors determine the utility for each possible behavior based upon two values configured for each task: priority and triggers. The priority of a behavior is simply its natural propensity to be the top behavior. Each behavior within a Task is assigned a priority, usually between 0.0 and 1.0, with higher value meaning the behavior is more important to have run. The second competing factor involves the triggers assigned to the behavior. Triggers determine if the behavior should run, represented as floating-point values.

The activation of a behavior, A, is represented as:

$$A = T * p$$

where T represents the set of triggers associated with the behavior and p representing the priority value for the behavior, with the result being a single numerical value. Triggers may have positive evidence or negative evidence (inverse) for a behavior to run. Triggers can be organized hierarchically and are logically combined as noted below:

$$\max\ (t_1, \dots, t_n) = \text{OR operator}$$
$$\min\ (t_1, \dots, t_n) = \text{AND operator}$$

Triggers are generated by modules outside of the behavior engine. Trigger examples include: ”*Is the character in a vehicle*?” or ”*Has the character been granted permission to engage threats?”*. Character perception produces many triggers including ones that note the presence of threats. Triggers can have any value between 0.0 and 1.0, creating more nuance in behavior activation than simply true/false.

Trigger values are stored in the TriggersManager, which essentially acts as a blackboard between the behavior engine and the modules that publish trigger values. The behavior engine will query the TriggersManager to ask for values when it goes to evaluate the activation level for behaviors within a Task. Publishing modules write to the TriggersManager with updated Trigger values. For example, when a new threat is detected, the CharacterMemory module will publish trigger values to indicate the threat, enabling the FindCover and EngageBehavior behaviors to run. Because those behaviors have been configured with a higher priority and are now activated, they will preempt the Move behavior and one of them will begin executing.

If the current behavior is replaced at the top of the priority queue by another behavior, then the current behavior is stopped and the new top behavior is started. Mechanisms exist to allow behaviors to latch on and prevent preemption, reducing behavior “jitter” when sensory inputs change.

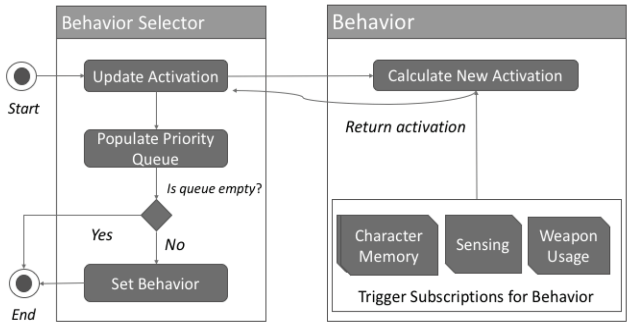

Figure 4: BehaviorSelector updating behavior activations

Behavior Selectors have a concept of termination behaviors. A termination behavior is one that, when complete, signals that the Task is done. Completion of the Task causes a signal to be sent to the originating Command to let it know that this character has completed the requested work, advancing to the next Command in the character/team’s plan. In Figure 3, the Breach behavior is the terminating behavior for the MechanicalBreachTask. When the breacher finishes that behavior, they signal the Breach command ended with success, at which point the overall command is complete. For the shooter characters executing CoverBreachTask, their Move behavior is their termination behavior, and when it completes they let the MechanicalBreach command know they are done.

Completion of a termination behavior does not mean that all behaviors stop. The behaviors can continue to compete so the characters can remain responsive. When a shooter indicates they have finished the Move behavior for a CoverBreachTask, the Move goes inactive and the other behaviors will compete until the MechanicalBreach command is considered done. That means these characters can engage threats that they detect even after they have completed the move to their cover location.

State Graph Behaviors

As was mentioned earlier, a Behavior is a package of related actions that accomplish a specific goal. The Behavior base class manages the state of a behavior, like whether it is currently running or has been initialized properly. It also stores the Triggers that it will use for computing its activation. The Behavior class includes calls to start work, stop work, and reset state.

Dante behaviors are currently all implemented as finite state machines. This allows reuse of behavior action states across many behaviors. StateGraphBehavior is the base class for these types of behaviors. This class holds the state machine mechanism for properly moving between states as those states complete their work. It also creates success and failure action states that derived behaviors can move to when they complete their work in either a success or failure condition.

Path Planning

Often the result of running a behavior is that the character moves from one location to another. This movement likely fulfills a purpose such as executing a command or reacting to something in the environment like finding cover from a threat. The path planning system in *Dante* extends the A* algorithm, providing a rich framework with which to generate paths that give variation to *Dante* movement behaviors.

The path planning system provides flexibility in the computation via two key mechanisms: costers and goal predicates. Costers define the shape that a path will take through the environment. For example, a character can define its path by only considering the distance to the goal meaning the path will be short and relatively direct. A character can use visibility avoidance costers to stay stealthy and generate paths that might take significantly longer to traverse but will keep the character out of sight. Other costers that can avoid tight spaces, vehicles, or even the potential firing lanes of friendly characters. Goal predicates define the goal location for paths. Usually, the goal location is known before planning begins, that is, it is a fixed location. With goal predicates, the goal becomes a function that must be satisfied.

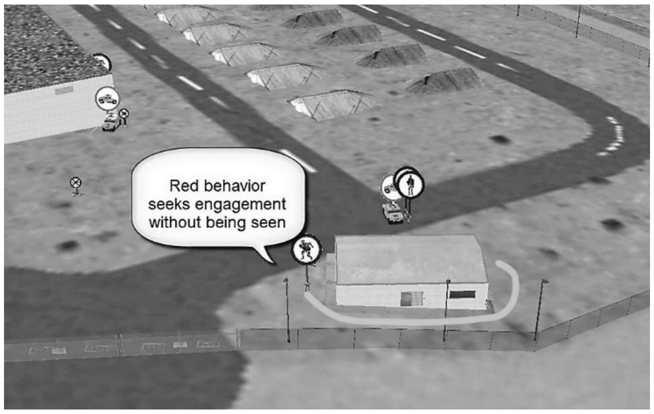

Figure 5: Character using goal predicate to find cover location

A key example in *Dante* is the act of finding cover to return fire. In Figure 5, the character does not know where to go, but dynamically searches for a location that will provide cover from a detected threat. Once the path is computed (showed as the line in Figure 5), it is handed to the mobility system. That system is responsible for moving the character along the computed path. It can accept a range of movement styles the character should use, such as running, walking, crawling, etc. It can also be configured with speed.

There are several behavior action states devoted to configuring path planning specifically to meet the needs of a given behavior. Other action states are devoted to working with the mobility system to move the character along those paths.

Results

Our goal for enabling non-technical personnel to quickly author *Dante* agents required devising a toolset for generating plans that red, blue, and/or neutral characters follow. The *Dante* Scenario Editor provides users the ability to import physical site models, including entities containing trigger publications to provide key information to the agents (e.g., intrusion sensors, vehicles, etc.). Users construct an agent's team plan by chaining together commands, with the completion of one command leading to the execution of the next. When the scenario is executed, this chain of commands is executed by the `RunManager`. As those commands are completed, it will determine whether the command completed successfully or not and then select the next command(s) to execute. The `RunManager` is also responsible for telling unassigned characters to go into the `Idle` command and for detecting when characters have been interrupted by a new command and properly notify their old command that they will not be completing it.

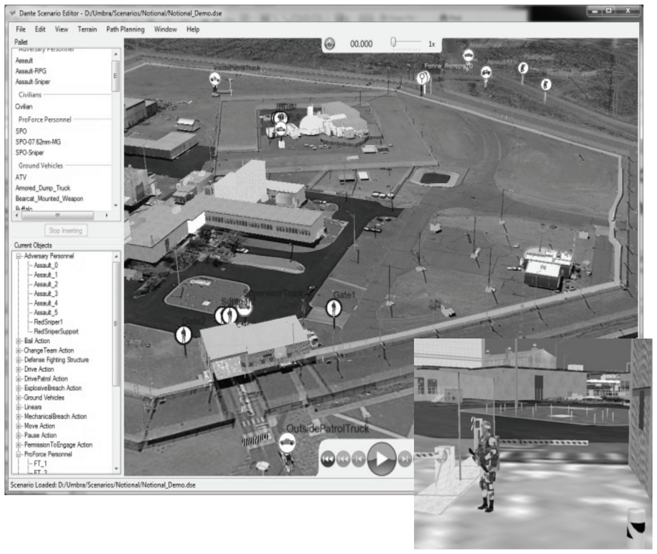
Dante Scenario Editor - D:/Umbra/Scenarios/Notional/Notional_Demo.dse
File Edit View Terrain Path Planning Window Help
Pallet
Current Objects
OutsidePatrolTruck
Scenario Loaded: D:/Umbra/Scenarios/Notional/Notional_Demo.dse

Figure 6: Dante Scenario Editor

Commands in *Dante* can be chained based on either success or failure. If a command completes successfully, then subsequent commands on the success path are signaled that they should consider running. If the command that just completed was the only command the next one was waiting for, then it will begin to execute. Commands, however, might have multiple input commands. If that's the case, then the next command could be configured to wait for all previous commands (AND) or just one (OR). If it should wait for all previous commands, then it will not execute until all previous commands have reported completion.

In the case when a command has failed, the `RunManager` will look for any chained commands that should be called on failure. This mechanism allows *Dante* scenario to branch when things are not going according to the original plan. There is a special mechanism in *Dante* called "fail forward." If there is not a failure branch for a failed command to follow, then it still signals the success branch actions. This allows the plan to continue forward, but likely it will encounter further problems that eventually result in a Terminate action. "Fail forward" aids users in not having to provide explicit detail how agents should respond if any portion of the plan fails, and safeguard against faulty user-defined plans where failed commands may not be necessary in accomplishing the scenario goal(s).

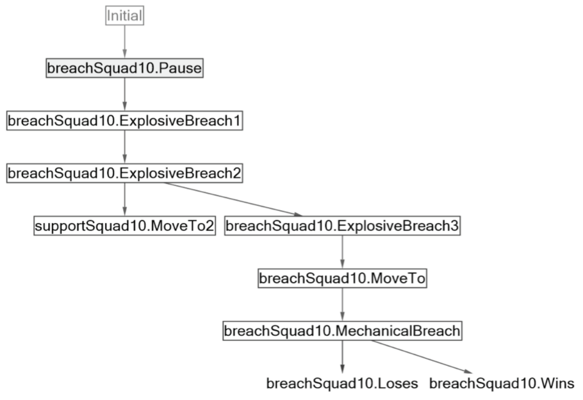
Initial
breachSquad10.Pause
breachSquad10.ExplosiveBreach1
breachSquad10.ExplosiveBreach2
supportSquad10.MoveTo2
breachSquad10.ExplosiveBreach3
breachSquad10.MoveTo
breachSquad10.MechanicalBreach
breachSquad10.Loses
breachSquad10.Wins

Figure 7: Example of a plan formed within Dante Scenario Editor

Figure 7 provides a snapshot from a *Dante* Scenario Editor displaying a red-team plan for breaking into a building within a secured facility, requiring the team to demolish multiple barriers between their start point and the building door. Each box represents a command issued to the team, with the yellow box indicating the current command being followed by the team. The arrows connecting the commands indicate what command(s) to run next if the command succeeds (green) or fails (red). In this case, the plan has not specified a failure branch for any command; the Terminate action is reached if the team cannot perform the `MechanicalBreach` to open the door. This example shows how commands may run concurrently, with the team splitting into two groups (breach & support) after succeeding in the second `ExplosiveBreach`.

Through *Dante* Scenario Editor, a user does not specify any details on how to achieve their strategic plans, leaving those decisions to the underlying architecture to determine each team member's role and handle the underlying behavior to successfully complete any issued command.

Future Research

The success of *Dante's* agent architecture stems from its ability to handle a wide range of situations and simplify the process of considering a wide variety of behaviors. However, they are limited by the experiences and/or imagination of the developer or analyst who designed it, ignoring potential behaviors that could expose new vulnerabilities. Furthermore, these systems require a degree of hand-tuning and subsequent user testing that makes it restrictive. Advances in planners, particularly path and route planning, also give the appearance of advanced behaviors and reasoning while being flexible. Planning-based AIs exist though are frequently driven by complex heuristics that require additional tweaking and tuning, are backed by yet another decision-making technology, or suffer from the overhead of re-planning should the inputs to the plan change. To address this drawback, subsequent research will explore applying evolutionary algorithms for generating novel behaviors, and comparing their success to expert-based behavior designs.

As capabilities for building new, more complex agents grow, new methods for validating the realism of these behaviors will be required. To test character behaviors, we suggest the option of using automated online experimentation to rapidly, and in a fully online manner, test character behavior with real human subjects. These subjects can be drawn from general online subject pools or from specific domains. The goal will be to rapidly evaluate the realism of character behavior in isolation – thus reducing the likelihood of character behavior deviating significantly from realism. Furthermore, these interactions can help train and adapt the behaviors.

Acknowledgments

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

Anderson, J., & Lebiere, C. (1998). *The Atomic Components ofThought.* New York, NY: Psychology Press.

Best, B., Lebiere, C., & Scarpinatto, K. C. (2002). Modeling synthetic opponents in MOUT training simulations using the ACT-R cognitive architecture. *11th Conference on Computer Generated Forces and Behavior Representation*, (p. 33).

Churchill, D., Saffidine, A., & Buro, M. (2012). Fast Heuristic Search for RTS Game Combat Scenarios. *Artificial Intelligence for Interactive Digital Entertainment*, (pp. 112-117).

Dill, K., & Mark, D. (2010). Improving AI Decision Modeling Through Utility Theory. *AI Summit at the Game Developers Conference.* San Francisco, CA.

Djordjevich, D., Xavier, P., Bernard, M., Whetzel, J., Glickman, M., & Verzi, S. (2008). Preparing for the aftermath: Using emotional agents in game-based training for disaster response. *IEEE Symposium on Computation Intelligence in Games*, (pp. 266-275). Perth, Australia.

Gottlieb, E., Harrigan, R., McDonald, M., Oppel, F., & Xavier, P. (2001). *The Umbra Simulation Framework.* Albuquerque, NM: Sandia National Laboratories.

Laird, J. (2001). Using a computer game to develop advanced AI. *Computer*, pp. 70-75.

Laird, J. (2012). The Soar Cognitive Architecture. MIT Press.

McPartland, M., & Gallagher, M. (2011). Reinforcement learning in first person shooter games. *IEEE Transactions on Computational Intelligence and AI in Games 3*, pp. 43-56.

Munoz-Avila, H., Aha, D., Jaidee, U., Klenk, M., & Molineaux, M. (2010). Applying Goal Driven Autonomy to a Team Shooter Game. *Florida Artificial Intelligence Research Society Conference*, (pp. 465-470).

Orkin, J. (2004). Applying goal oriented action planning to games. AI Programming Wisdom 2, pp. 217-227.