Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

INTERACTIVE MULTI-LEVEL VIRTUAL TACTICAL SIMULATION: THE DEVELOPMENT OF AN AUTONOMY ARCHITECTURE TO IMPROVE TRAINING EXPERIENCE

Edison P. de Freitas[1,2], Eliakim Zacarias[2], Raul C. Nunes[2,3], and Luis A. L. Silva[2,3]

[1]Department of Applied Informatics, Federal University of Rio Grande do Sul, Porto Alegre, RS, BRAZIL
[2]SIS-ASTROS GMF Project, Federal University of Santa Maria, Santa Maria, RS, BRAZIL
[3]Department of Applied Computing, Federal University of Santa Maria, Santa Maria, RS, BRAZIL

ABSTRACT

Multi-level simulation setups are crucial to train individuals with different learning needs in the same virtual environment. For users at the group command level in virtual tactical simulations for defense/military, tasks simulated at the unit (lower) levels are usually executed autonomously. The contribution of this paper is to detail how these autonomous agent tasks are performed in a chain of actions and commands that include the possibility of high-rank users' interaction with the simulations. The work reports on the experience of designing and implementing an autonomy architecture for an agent-based simulation system called SIS-ASTROS. In this system, autonomous tasks executed by simulated artillery batteries are grounded on the exposition and controlled execution of the API of the SIS-ASTROS simulator. In a user-friendly specification, simulation analysts can provide scripts for doctrine-based agent behaviors, where an autonomy simulator module executes these scripts to improve the overall training experience.

1 INTRODUCTION

Agent-based simulation (ABS) (Macal 2016) is a technique of great importance to support the development of training activities in diverse domains (Abar et al. 2017), from healthcare (Lee et al. 2020) to industrial maintenance (Bordegoni and Ferrise 2023). A domain in which ABS setups prove their value is the defense/military (Hill and Miller 2017). Simulation is particularly valuable in the military due to the high risk and costs involved in real-world training activities. However, military training has particular characteristics that should be observed to enhance the trainees' skills effectively. Particularly, the problem investigated in this paper is the development of autonomous components to support the project and implementation of multi-level aspects of military training in a single simulation system/environment.

Multi-level military training is characterized by merging different levels of training skills and trainees into the same training setup (synthetic environment). Simulation setups used for this type of training usually blend the Live, Virtual, and Constructive conceptual simulation types defined by the Department of Defence of the United States (DoD) (DoD 2013). In many situations, however, there is a need to support another degree of multi-level simulation refinement, even when implementing just one of these conceptual types. This happens in a virtual tactical simulation, in which constructive aspects are considered in contrast to a pure virtual simulation, which opens the possibility to approach different levels of decision-making that simultaneously target the training of military officers of different ranks using the same simulation environment. This is the case of SIS-ASTROS (Pozzer et al. 2022), which is a virtual tactical simulation environment for the Artillery Saturation Rocket System for Area Saturation (ASTROS) (AVIBRAS 2019).

The multi-level training setup analyzed in this work aims to support the training of higher-rank officers without requiring other users to be trained at the lower levels at the same simulation exercises. This is a need because when training a group commander, the tasks executed by the batteries that compose the group still have to be performed in the simulations. However, the simulation setup does not always have the battery commanders being trained in those simulation exercises, and sometimes only some of them may participate in the simulation-based training activities. Thus, it is required that the modeled battery agents autonomously execute their tasks so that the higher-rank officer can proceed with its artillery group training tasks in the simulation setup.

Regarding the above-described problems, the approach developed in the proposed solution advances state-of-the-art development of ABS systems through the following contributions. It starts by proposing the exposition of the Simulator Application Program Interface (API) so that high-level script commands are used to model doctrine-based task-level autonomous agent behaviors in the simulations. Then, unlike most fully autonomous agent implementations developed in simulation systems and computer games, the work relies on an interactive autonomous behavior execution, which allows training users to interact with the autonomous agent actions unfolding in the simulations. The development of autonomous agent support detailed in this work considers that the executed tasks must follow the military doctrine as closely as possible to meet the required educational goals. This work also presents details of the project and implementation of an autonomy module in charge of interpreting and executing the agent behaviors in the simulations. A case study of implementing the proposed solution in the SIS-ASTROS simulator is provided to show how such an autonomy architecture can be reused in other simulation system development projects.

The reminding of this paper is organized as follows: Section 2 presented a background review to this work. Section 3 describes the proposed solution to enhance the interaction in multi-level virtual tactical simulation, while Section 4 provides a running example of this solution implemented in SIS-ASTROS. Section 5 presents the lessons learned and a prospective discussion, while Section 6 concludes the paper.

2 BACKGROUND TO THIS WORK

There are relevant issues regarding the automation of agents' behaviors for ABS systems (Macal 2016). The first is that automated behaviors, usually modeled by different Machine Learning (ML) techniques, typically focus on implementing intelligent *local* behaviors for agents in the virtual environments, similar to what is usually found in computer games (Galway et al. 2008). These Artificial Intelligence (AI) behaviors include path planning, avoiding static and mobile obstacles, moving and organizing in different formations, selecting and organizing positions on the terrain, and different interaction tasks. Despite the most frequent use of local agent behaviors implemented by various AI techniques, there is a need to combine them to model higher-level task behaviors for groups of units, such as when modeling and executing more complex tactical tasks of selection, recognition, and occupation of given tactical areas with artillery battery groups in the SIS-ASTROS simulator.

Autonomously executing high-level strategic and tactical tasks in many systems requires exploring planning techniques from AI. This is the case of autonomous agent behaviors implemented in many Real-Time Strategy (RTS) games (Ontañón et al. 2013). In the Starcraft game, a known battle game, behaviors of high-level task agents are modeled and executed as micromanagement scripting techniques (Rogers and Skabar 2014). There, the script commands combine and execute the local agent behaviors in the game environment so that high-level goals are achieved in the game. Another worth-mentioning instance of such computer game implementation in which the system API is exposed so that different AI algorithms are used is the Spore (Schrader et al. 2016). In a single game scenario, this multi-level computer game enables the user to play at the individual level of a simulated entity, controlling just that entity, or at the group level, where some entities are autonomous. This change in the levels of command progressively appears while the user passes the game stages in an evolutionary story. However, even in upper levels of game interaction, where the user controls groups of entities, the game allows the user to control only a specific entity or one group. In a medieval scenario, another computer game with similar characteristics is "Mount and Blade 2: Bannerlord" (Ricardo 2021). This game allows the user to control a single agent and groups of agents according to the different levels of commands. However, users can switch levels during gameplay whenever necessary. Proportionally speaking, these game features have similarities with the problem investigated in this work. As the agent-based simulation environments that we investigate, Spore (Poli et al. 2012) and "Mount and Blade 2: Bannerlord" (Majewski et al. 2023) could also be used for educational purposes.

Another problem is that, in most system implementations, these autonomous task solutions execute the role procedures from the beginning to the end without the possibility of human intervention (Synnaeve et al. 2016). This is a problem with the intended use of agent-based simulations for training. In the real world, there is a line of communication between the multi-unit group and the single-unit commanders regarding the execution of given missions. In the simulations for training tactical procedures in the military, it is important that the higher-rank officers, even if the focus of their training is not the command and control of lower-level agent tasks, are able to interact more frequently with these autonomous behavior executions. There are other cases in which this possibility is desired, such as when simulation users have to interfere with the autonomous executions to ensure the proper execution of certain agent behavior in the simulations, avoiding a military simulation problem that can issue a simulation error when the autonomous agent behavior is executed or that deviates from the underlying military doctrine executed, for instance.

Connected to the above-mentioned problems of implementing autonomous behaviors that usually execute to the end without the possibility of users' interference, these behaviors are usually hard-coded in the implemented simulation systems. Besides the lack of flexibility and all software engineering-related problems in this approach, it also hinders the possibility of updating these behavior models when necessary. In (Synnaeve et al. 2016), the authors present a library that exposes the functionalities of the Starcraf-like games to allow their usage by ML algorithms. This approach partially solves the software engineering problems presented by the hard-coded approach in computer games. Still, it does not address the need for human intervention in simulation-based training.

To sum up, the multi-level architecture of this work is focused on a strategic and micromanagement task allocation approach for RTS games detailed in (Rogers and Skabar 2014). This framework bridges the gap between strategy and unit-level micromanagement to address the challenges of controlling individual entities and groups of these entities in the same virtual environment using AI algorithms. Despite their proposed architecture, that framework does not support the desired level of user interactions required by simulation systems for educational/training purposes, which is the goal of this current work.

3 THE AUTONOMY ARCHITECTURE DESIGN

3.1 Architecture Overview

To perform simulations of groups of agents, the ASTROS group in this work, the high-rank simulation system users are not concerned with the low-level specific tasks of each individual agent that compose a group. Conversely, their training concerns the entire set of agents as an ensemble. The low-level agent compositions must perform low-level tasks autonomously to achieve this goal. In these tactical battery simulation exercises, for instance, each battery is commanded by another user at a lower rank. However, this is not the case for all simulations, i.e., in many occasions, one, two, or even all batteries have no commander involved in the current simulation exercises, i.e., a user being trained as a battery commander, as schematically represented in Figure 1. Still, their tasks need to be performed to allow the training of the higher-rank officers commanding the artillery group.

In the above simulation scenario, the batteries need to perform their tasks autonomously instead of waiting for commands from a user. This resembles the usage of constructive simulators. However, these tactical simulations have important differences from these constructive simulations. In this case, the training users should have the opportunity to closely follow the execution of the low-level tasks, or to interfere in their execution whenever the course of action requires doing so. Even though the ultimate goal is not directly to interact with the simulated low-level battery tasks, they are important for the training methodology of users at the group level, i.e., these users are interested in following the unfolding of these tasks because what happens in this lower level may impact their decisions. This differs from the logic of almost always fully autonomous agent-based constructive simulators, which completely abstract the low-level tasks, working exclusively at the coarse-grained level.

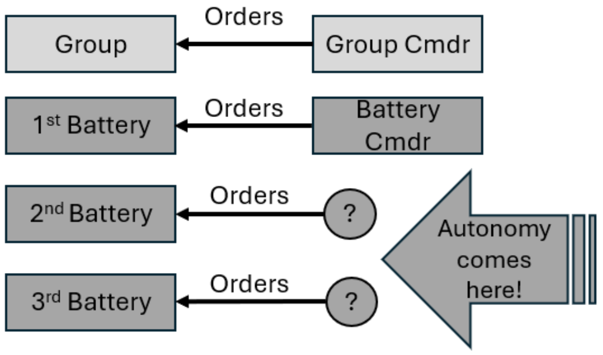


Figure 1: Simulated group and batteries with their (possible) users.

In this project, a dedicated and reusable module in the simulator was designed to implement the required autonomy support. This module is isolated from the rest of the simulator and interacts with it through a communication layer. Figure 2 represents the main building components of this module and its interaction with the simulator. This proposal is based on executing external and user-friendly scripts (i.e., for simulation analysts) fed to an interpreter. The output of this interpreter is the set of simulation actions and commands that will govern the execution of a Finite State Machine (FSM). According to the exposed simulator API, the FSM executes the given autonomous behaviors. This FSM sends simulation requests with the identified tasks that the entities in the simulator have to execute. At the same time, it saves the simulation states in particular variables so that the module can implement functions to recover in case of unexpected simulated problems and unwanted and not-realistic behaviors or upon a request from the user to change to course of simulation actions or to stop the autonomous execution.

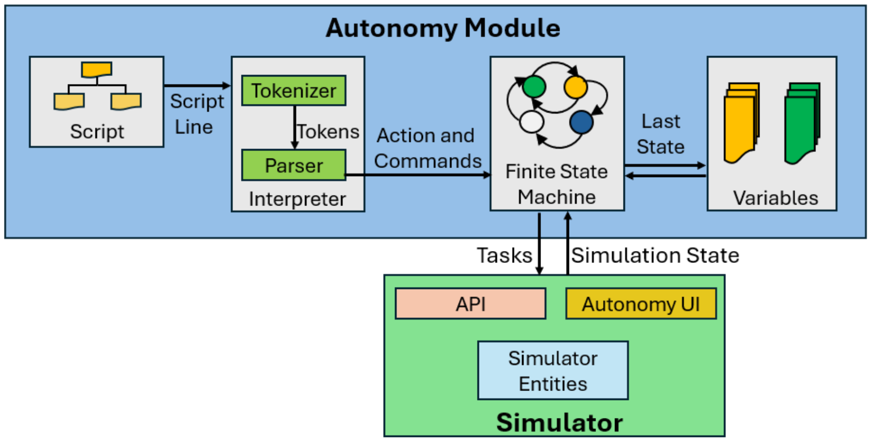


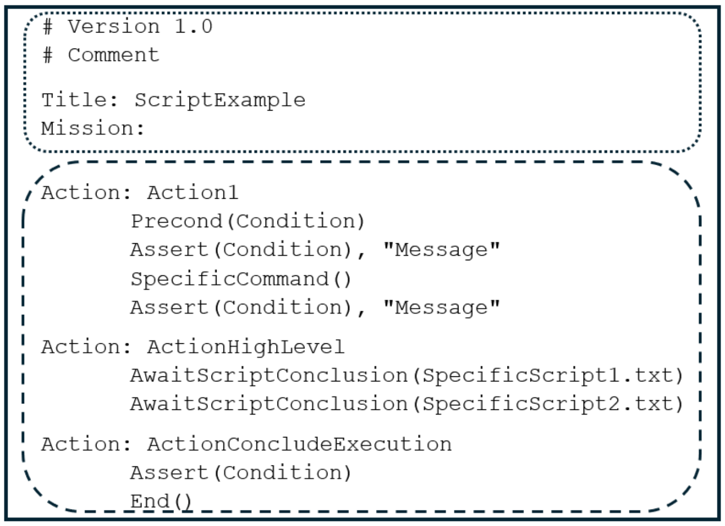
Figure 2: Autonomy module building components and its interaction with the simulator.

Unlike most fully autonomous agent behaviors implemented by standard ML algorithms in simulation systems and computer games, an important aspect to be highlighted is that the autonomy simulation system module implemented in this project follows strict rules of the military doctrine. Another relevant aspect is the flexibility the proposed solution provides in making it possible to switch back and forth between the autonomous and the manual command of the low-level simulated units (i.e., the batteries in the simulation system). This is a crucial feature to allow the desired interaction to support improving the training experience for the users.

3.2 Architectural Components

Scripts: The autonomy scripts are detailed in a simulation analyst (user-friendly) language stored as multiple command lines in text files, not hard-coded in the simulator as in most autonomous simulation system implementations detailed in the literature. These files are written according to a specific syntax interpreted by the module. The scripts are organized in a hierarchical scheme in which specific agent behaviors are described. This hierarchical organization allows their combination according to the needs of a given situation in which autonomy execution is required.

Figure 3 presents a template to exemplify how a script is organized. The dotted line box at the top highlights the script's header, in which, besides the title, the simulation analyst can provide information about the script version. The header ends with the reserved word *Mission*, which states that the script's content starts in the following. The dashed line box contains the script's content. The detailed doctrine-based tactical tasks in this script content are organized as *Actions*, composed of *Commands* or calls to other more specific scripts. Commands often represent direct simulator executions whenever they can be mapped to specific commands in the exposed simulator API. However, these commands can also be broken into more detailed scripts. In the template presented in Figure 3, *SpecificCommand* represents a command that has to be executed to perform the agent behavior in the simulation described by *Action*1. In action *ActionHighLevel*, two calls to specific scripts can be observed using the directive *AwaitScriptConclusion*. Besides the commands and the call to specific scripts, the *Precond* and *Assert* commands are also required in this scripted plan specification. The first represents preconditions that have to be satisfied in the simulations before allowing the execution of a given command. For instance, a vehicle cannot leave a given tactical location if its crew is not onboard. The second represents the execution of validations (sometimes implemented as post-conditions in plan-like specifications) that need to be performed once a given command is executed in the simulations. They may raise unwanted simulations interrupting the simulation execution while waiting for the user's intervention. The issues requiring training users' interventions may be simple *Warnings*, which inform the users about a simulation situation that they may or may not interfere, or *Fatal*, which compulsory demands the user's intervention in the autonomous executions. This mechanism maintains the realistic consistency and execution flow of the simulator.



```
# Version 1.0
# Comment

Title: ScriptExample
Mission:

Action: Action1
        Precond(Condition)
        Assert(Condition), "Message"
        SpecificCommand()
        Assert(Condition), "Message"

Action: ActionHighLevel
        AwaitScriptConclusion(SpecificScript1.txt)
        AwaitScriptConclusion(SpecificScript2.txt)

Action: ActionConcludeExecution
        Assert(Condition)
        End()
```

Figure 3: Template of the actions and commands detailed in a script externally specified and loaded by the autonomy module whenever an autonomous task behavior is executed in the simulator.

Interpreter: The interpreter is a major component that translates the script lines into commands and actions used by the FSM. It is composed of two internal sub-components, the Tokenizer and the Parser.

The first is responsible for taking the script lines and extracting from them the tokens that the Parser will process. Then, the second maps these tokens to the commands and actions of the exposed simulator API.

Finite State Machine: The FSM controls the execution flow of the autonomously simulated tasks and the actions and commands sent to the simulation entities. It also saves the current simulation status, the last state that the entities executed the ordered actions and commands. It also updates itself by reading the simulation state. As required during the development of our project, since the simulator was being improved on different fronts while the autonomy module was being implemented, the implemented FSM was crafted to include new states according to the increment and adjustment of the simulator's functionalities as required by the simulation system users.

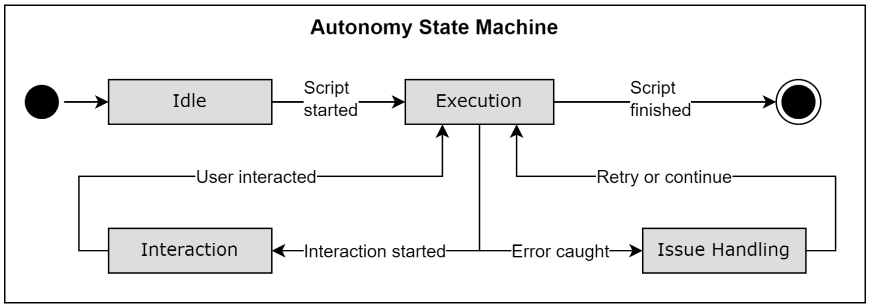


Figure 4: The control of the autonomous task behaviors in the simulations is developed by a designed Finite State Machine component in the autonomy module.

Figure 4 presents the designed FSM. The first state *Idle* represents when the battery is ready to become autonomous. Once the script starts, it moves to the second *Execution* state, representing the execution of the autonomous behaviors described in the upload scripts. The script specifications may run until the end, and then the FSM moves from the *Execution* to the end state. However, this execution can be interrupted by user interaction which moves to FSM to the *Interaction* state or the *IssueHandling* state upon unwanted simulation situations. Eventually, it returns to the *Execution*, either after the user interaction or the user resolution of the simulation issues, until the script is finished.

Variables: This component plays a very important task in the interactive autonomy capabilities to support the development of training activities in the agent-based simulations. Among other aspects, it is responsible for saving the last state of the FSM before sending the commands and actions to the simulator entities. As experienced in SIS-ASTROS project, this is crucial as this memory allows restoring a previous state in case the simulations need to be changed due to the occurrence of an unwanted and unrealistic simulation situation and, therefore, there is a need for correction, which will demand the intervention of the user. Moreover, this restoration is required in case the user, due to different reasons, decides on a specific part of the execution of the autonomy behavior, allowing the user to manually execute the simulated operation. Thus, it means that the designed autonomy mechanism permits the agent to follow a pre-specified course of actions and commands and to switch back and forth between simulated executions that training users manually control. An example is when the module selects an area to park the vehicles that compose the battery. For a given reason, the user may want a given battery deployed in a specific location in the tactical scenario. Hence, instead of leaving the autonomy module to decide the tactical locations where the battery vehicles should occupy, the user interacts in that specific part of the execution and performs this tactical course of action by himself/herself.

Simulator API: As designed in this project, the autonomy module is separated from the rest of the simulator as much as possible. However, moving forward while the autonomous behaviors unfold requires access to the user simulation functionalities and the different elements that compose a battery represented in the simulator, i.e., the virtual representations of the vehicles and the military staff inserted in the virtual scenario. Thus, the autonomy module can send commands to them in the simulator. In a project issue that is not simple to design and implement, as experienced in this project, the simulator has its API detailed and exposed through this component. Therefore, the simulator API receives tasks sent by the FSM in the autonomy module and provides control of the simulator entities.

Simulator Entities: The simulator entities represent the various different agents represented in the simulator. These agents are dynamically organized at various levels of detail in the simulation system, from individual units to groups of units. These tactical organizations can be formed whenever high-level simulation tasks are executed. For instance, recognition groups can be formed whenever tactical position recognition procedures have to be developed to complete a given battery mission.

Autonomy UI: One of the contributions of the proposal presented in this work is identifying the need for different degrees of user interaction while developing autonomy task behaviors in the simulations. This is possible through this UI component of the autonomy module. Among other functionalities, it provides high-level information and contextual feedback for the training users about the current status of the autonomous executions of the intended tasks. For training purposes, information is provided at the task execution level, where the main task activities are summarized, and their execution status is presented to high-rank training users (e.g., battery group commanders). However, this user simulation interface also shows the actions under execution, which permits users to follow the lower-level agent simulations. Moreover, it fundamentally provides an interface through which the training users can interfere in executing the autonomous behaviors according to their simulation needs.

4 A CASE STUDY FOR THE INTERACTIVE MULTI-LEVEL SIMULATION

This section presents a simulated scenario in which the autonomy module controls a battery so that their autonomous behaviors are executed along with circumstances that require the intervention of the user. Considering a training set in which a group commander needs to perform a given mission in a tactical area and assuming that three batteries are configured to run autonomously, the autonomy module takes control once the group commander assigns a given mission to each one of the batteries. Figure 5(a) presents this overall scenario with three batteries, each one inside its own rectangular area of operation.

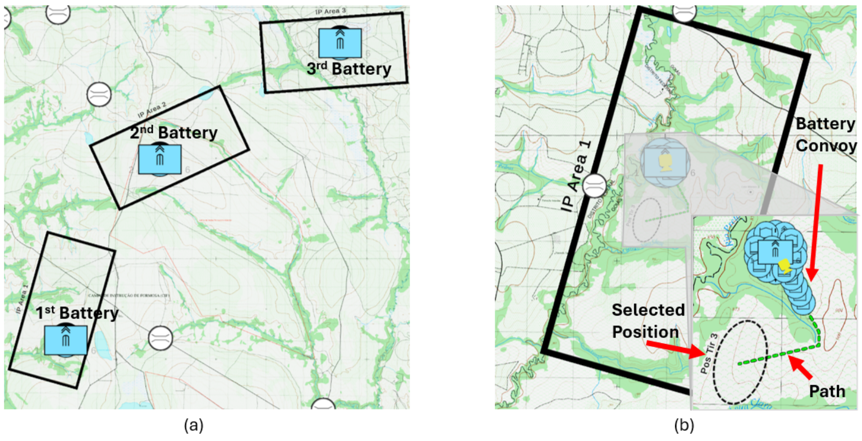
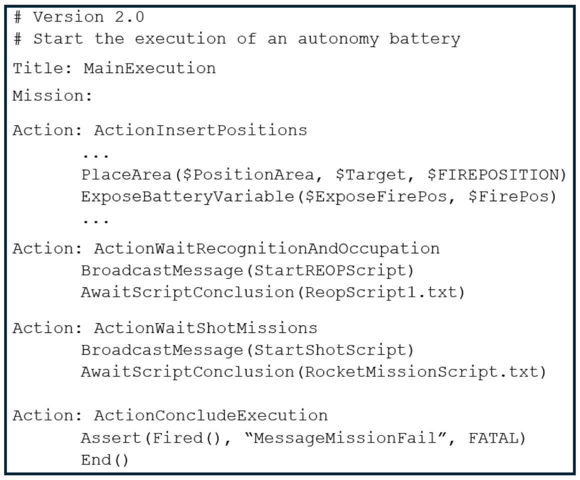


Figure 5: Screenshot of the SIS-ASTROS simulator presenting the tactical maneuver in the 2D map interface: a) Overview of the operation scenario showing the three batteries. b) Focus on one of the batteries showing its automatically computed path towards the selected position.

The first activity that must be executed is selecting the tactical positions to be occupied by the vehicles and the members of each battery during the mission execution. Each battery executes its own instance of the autonomous navigation behaviors implemented in the SIS-ASTROS simulator (Brondani et al. 2017). These low-level AI behaviors are conducted in the simulations according to task movement commands interpreted by the autonomy module. Then, once the positions are selected, the units of each battery move toward them, using the AI pathfinding solution developed for the SIS-ASTROS (Chagas et al. 2022), (Figure 5(b) - with a highlight to the zoomed-in part on the right showing the convoy starting its movements along the path towards the selected position). Following the sequence of actions and commands detailed in the task specification as represented in the executing scripts, the positions are occupied according to the doctrine-based agent behavior implementations required by the given mission. Finally, the battery shots are executed given the shot order emitted in the simulator by the training users, and the group commander is informed if the mission was accomplished.

Figure 6(a) presents the script that starts the execution of an autonomy battery. Only parts of the script relevant to understanding the situation are presented. Considering the maneuver situation, the first action is to autonomously select the tactical positions used in the mission. Several script commands execute this action. The first *PlaceArea* command is responsible for selecting a given position in the mission area that will be used for the given activity. In this case, the example shows the selection of the fire position. After selecting a given position as a *FirePosition*, the variable that defines and memorizes this position is exposed to the autonomy module for further usage by the *ExposeBatteryVariable* command.

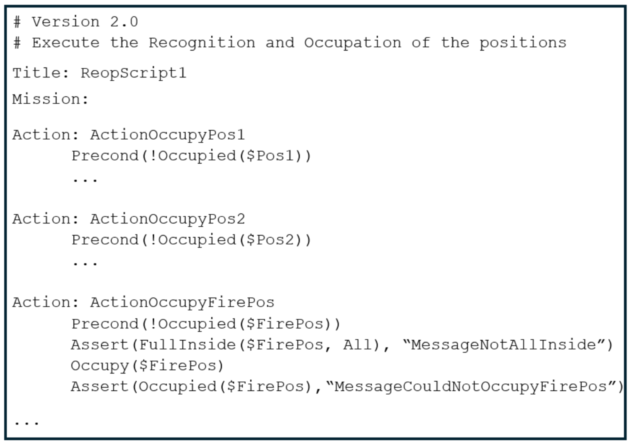


```
# Version 2.0
# Start the execution of an autonomy battery
Title: MainExecution
Mission:

Action: ActionInsertPositions
       ...
       PlaceArea($PositionArea, $Target, $FIREPOSITION)
       ExposeBatteryVariable($ExposeFirePos, $FirePos)
       ...
Action: ActionWaitRecognitionAndOccupation
       BroadcastMessage(StartREOPScript)
       AwaitScriptConclusion(ReopScript1.txt)

Action: ActionWaitShotMissions
       BroadcastMessage(StartShotScript)
       AwaitScriptConclusion(RocketMissionScript.txt)

Action: ActionConcludeExecution
       Assert(Fired(), “MessageMissionFail”, FATAL)
       End()
```



```
# Version 2.0
# Execute the Recognition and Occupation of the positions
Title: ReopScript1
Mission:

Action: ActionOccupyPos1
      Precond(!Occupied($Pos1))
      ...

Action: ActionOccupyPos2
      Precond(!Occupied($Pos2))
      ...

Action: ActionOccupyFirePos
      Precond(!Occupied($FirePos))
      Assert(FullInside($FirePos, All), “MessageNotAllInside”)
      Occupy($FirePos)
      Assert(Occupied($FirePos),“MessageCouldNotOccupyFirePos”)
...
```

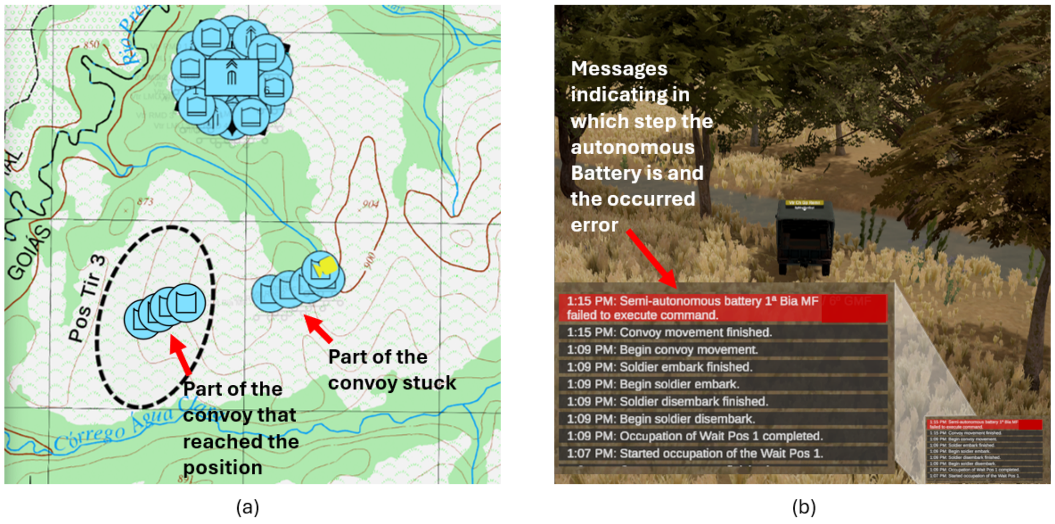
(a) (b)

Figure 6: (a) Script that starts the execution of an autonomous battery. (b) Subscript responsible for executing the recognition and occupation of the selected tactical positions.

The following actions in the script depicted in Figure 6(a) perform the following steps in the mission execution. The *ActionWaitRecognitionAndOccupation* action first sends a message that will be displayed in the simulator interface so that the training user is informed of the activities in which the execution currently is, and the *BroadcastMessage* command describes it. Once the user can follow the autonomous executions, it may interfere with it. For instance, these users can pause the execution to check if the autonomy is performing the actions that favor their educational goals. Then, a subscript is called by the *AwaitScriptConclusion* command. This subscript is responsible for executing the battery behaviors regarding the recognition and occupation of the tactical positions. Figure 6(b) presents a sample of this subscript. The next action executes the battery shots, showing its execution to simulation users and calling another subscript responsible for the specific shooting commands. Finally, the *ActionConcludeExecution* action verifies, by the *Assert* command, if the mission is accomplished as intended. If it is not, it displays a message and raises a simulation issue/error. In this example, a "FATAL" error is raised, which will demand the user intervention.

Figure 6(b) presents part of the subscript that executes the occupation of the selected tactical positions. Notice that at the beginning of each action, there is a *Precond* clause to be satisfied. For all three actions in this example, the pre-condition checks if that selected terrain position is not already occupied. In the third

ActionOccupyFirePos action, it is also possible to observe two *Assert* clauses. The first validates that all vehicles are in the desired places and ready to occupy the tactical position effectively. At this step, there is a possibility for a new user intervention. Once the vehicles are moving toward the *FirePos*, the AI path planning algorithm may select a path to be used by the battery. This itinerary may be muddy, where part of the convoy can become stuck and unable to proceed toward the destination position. This is the situation in Figure 7. This simulated military problem is used to train low-rank officers commanding a battery. Even though this situation is not intended to be part of the training of high-rank officers commanding groups, they must be aware of this real-world situation. Moreover, as the automation module uses the same simulator API as the low-rank end-users, the autonomously executed agent implementations may face some of the same problems. Thus, there is a need for human intervention, which is executed by the high-rank end-user through the Autonomy UI as presented in Figure 8(a). In this interface, the user can take control of the autonomous battery, select an appropriate path to resolve the simulated problem as presented in Figure 8(b), and give the control back to the Automation Module to continue the execution from this step until it finishes the action as presented in Figure 8(c). After executing the *Occupy* command, at the end of this part of the subscript, another *Assert* is executed to verify it. In case of any problem, a simulation message is raised, alerting the user about the problem and again opening an opportunity for user intervention.


Pos Tir 3
Part of the convoy that reached the position
Part of the convoy stuck
GOIÁS
Córrego Água Clara
(a)
Messages indicating in which step the autonomous Battery is and the occurred error
(b)

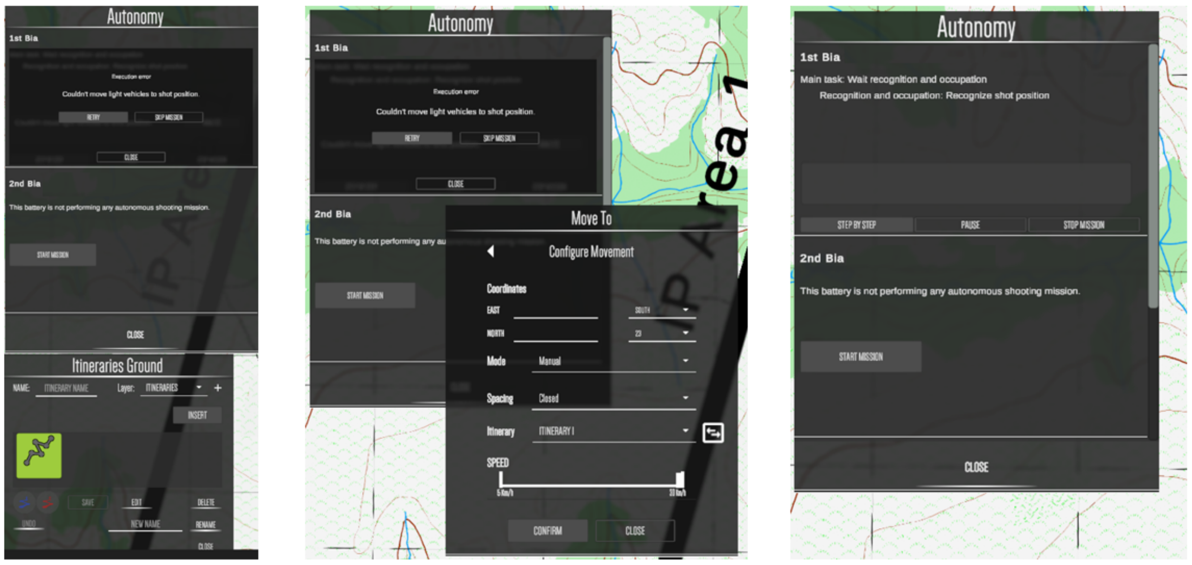
1:15 PM: Semi-autonomous battery 1ª Bia MF / 6º GMF failed to execute command.
1:15 PM: Convoy movement finished.
1:09 PM: Begin convoy movement.
1:09 PM: Soldier embark finished.
1:09 PM: Begin soldier embark.
1:09 PM: Soldier disembark finished.
1:09 PM: Begin soldier disembark.
1:09 PM: Occupation of Wait Pos 1 completed.
1:07 PM: Started occupation of the Wait Pos 1.

Figure 7: Screenshot of the SIS-ASTROS simulator presenting the operation map in the 2D and the 3D interfaces: a) Map presenting part of the convoy stuck on its way to the selected position. b) The 3D interface presenting one of the convoy vehicles stuck.

5 DISCUSSION - LESSONS LEARNED

The module that implements the autonomy battery task behaviors was developed in parallel with the implementation of new simulation functionalities on the SIS-ASTROS simulator. Many parts of the autonomy code were refactored during this process to comply with simulator changes. This directly impacted the project and implementation of the autonomy module, as it was designed to call the simulator through its exposed API. To overcome this problem, we started implementing a simple set of battery activities. Even knowing that the mentioned necessary simulator changes could impact the autonomy module development, this project decision was taken so that the development team could gain knowledge on how to build such a module, and the team could benefit from this experience in the remaining implementation process.

A specific case worth mentioning is the development of the scripts that support the autonomy module. Initially, the script for autonomous battery behaviors was written in a simplified specification line. One problem with this approach was the lack of support for repeating required autonomous battery tasks, such



Execution error

Couldn't move light vehicles to shot position.

This battery is not performing any autonomous shooting mission.

Main task: Wait recognition and occupation

Recognition and occupation: Recognize shot position

Figure 8: Autonomy UI: a) The user takes control of the autonomous battery. b) The user manually sets a new path, waits for its execution, and returns the control to the Autonomous Module. c) The Autonomy UI shows that the action was completed.

as fire commands. In the simplified version, these commands were detailed to perform just once in the simulations. Another problem with this simplified script description was the lack of parallelism as required by the underlying military doctrine detailed in the script actions and commands, i.e., operations that occur simultaneously in the real world. Later in the autonomy module implementation, this was changed by separating the script into a set of specification files. Then, specific directives introduced in the script specification were used to run iterative and simultaneous actions and commands. Another relevant aspect of this change to multiple descriptions was how the scripts were organized. A hierarchical organization was adopted so that higher-level scripts called others with more specific behavior descriptions until the desired autonomy battery task behavior was completely fulfilled.

As the autonomy module uses the same simulator API used by the training users, there are cases in which inconsistencies may occur during the execution of specific script actions and commands. When dealing with the training of lower-rank officers, i.e. the battery commanders, these simulation problems are desired as the simulator has to allow the users to make simulation mistakes, most of the time mistakes related to the incorrect exploration of the military doctrine. This simulated try-and-error process is central to providing experience and supporting learning by post-action review. However, when the training of higher-rank officers is concerned, using the same simulator API may lead the simulator execution to inconsistent states. Thus, it was necessary to develop an autonomy module solution to guarantee consistency or to call the user's attention to a given situation that might require his/her intervention. For instance, this solution was mostly implemented by the validations provided by the *assertions* statements in the scripts.

Another important improvement for the autonomous implementations was the synchronization support to execute the simulated battery actions. Taking as an example the shooting execution, in the first version of the developed autonomy module, they fired once the batteries were ready to shoot. According to the military doctrine, this is just one of the possible cases. Other possibilities are a pre-defined timestamp to fire and an $on-demand$ order that waits for a specific shooting command. Hence, it was necessary to implement the time support to synchronize the autonomy module executions with the simulation time so that the autonomous battery could act according to the applicable doctrine. Figure 9 presents the structure of the preliminary implementation and the improved (synchronized) one.

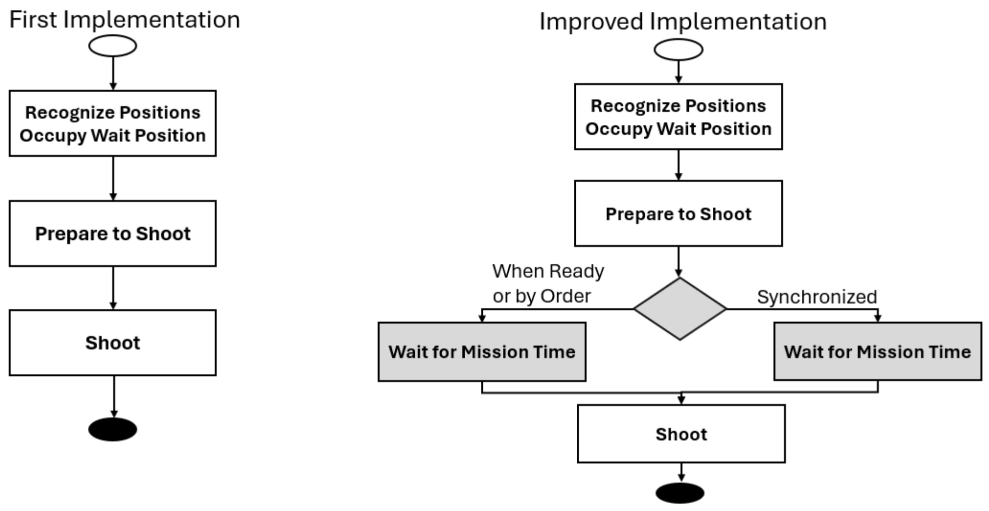


Figure 9: The preliminary and the improved (synchronized) version of the firing order.

6 CONCLUSION AND FUTURE WORK

This paper discusses the design and implementation characteristics of an autonomy architecture. The aim was to report a research and development experience that can be reused by others in need to develop simulators with interactive multi-level simulation setups to improve users' training experience.

To enable the desired level of interaction, the first action was exposing the simulator API so that high-level script commands could be used to model task-level autonomous agent behaviors. Therefore, simulated entities could act according to the designed behaviors. However, exposing the API could not fulfill the requirements for an improved multi-level simulation experience. In the SIS-ASTROS simulator, it was necessary to design a solution that allowed the interaction of the simulation users with the simulated entities during the execution of the autonomous behaviors. This was required even at a higher hierarchical position, the higher-rank officers usually interact with the lower-level ones in specific decisions of the batteries' missions. As there is no human commanding the autonomous batteries, the higher-rank officers need to have the possibility to interact with these batteries directly to solve military simulated problems.

Future work in developing the designed autonomous support for the SIS-ASTROS simulator points to directions of using ML techniques to learn the behaviors of end users when they interact with the autonomous execution. That is particularly relevant in the occurrence of simulated problems so that, under the same conditions, the Autonomous Module solves the problems by itself according to the military doctrine and the current battle situation, increasing the provided level of autonomy.

ACKNOWLEDGMENTS

We thank the Brazilian Army Strategic Program ASTROS for financial support through the SIS-ASTROS GMF project (898347/2020) - TED 20-EME-003-00.

REFERENCES

Abar, S., G. K. Theodoropoulos, P. Lemarinier, and G. M. O'Hare. 2017. "Agent Based Modelling and Simulation tools: A review of the state-of-art software". *Computer Science Review* 24:13–33.

AVIBRAS 2019. "Artillery Saturation Rocket System". https://www.avibras.com.br/, accessed 15th August 2024.

Bordegoni, M. and F. Ferrise. 2023, 05. "Exploring the Intersection of Metaverse, Digital Twins, and Artificial Intelligence in Training and Maintenance". *Journal of Computing and Information Science in Engineering* 23(6):060806.

Brondani, J. R., E. P. de Freitas, and L. A. Silva. 2017. "A task-oriented and parameterized (semi) autonomous navigation framework for the development of simulation systems". *Procedia Computer Science* 112:534–543.

Chagas, C., E. Zacarias, L. A. de Lima Silva, and E. Pignaton de Freitas. 2022. "Hierarchical and smoothed topographic path planning for large-scale virtual simulation environments". *Expert Systems with Applications* 189:116061.

DoD 2013. *Department of Defense Modeling and Simulation (M&S) Glossary (DoD 5000.59-M)*. CreateSpace Independent Publishing Platform. https://apps.dtic.mil/sti/pdfs/ADA39800.pdf, accessed 15th August 2024.

Galway, L., D. Charles, and M. Black. 2008. "Machine learning in digital games: a survey". *Artificial Intelligence Review* 29(2):123–161.

Hill, R. R. and J. O. Miller. 2017, Dec. "A history of United States military simulation". In *2017 Winter Simulation Conference (WSC)*, 346–364. doi: 10.1109/WSC.2017.8247799.

Lee, R., N. Raison, W. Y. Lau, A. Aydin, P. Dasgupta, K. Ahmed *et al*. 2020, Oct. "A systematic review of simulation-based training tools for technical and non-technical skills in ophthalmology". *Eye* 34(10):1737–1759.

Macal, C. 2016. "Everything you need to know about agent-based modelling and simulation". *Journal of Simulation* 10:144–156.

Majewski, J., M. Mochocki, and P. Schreiber. 2023. "Pushing in from the Margins: Player Efforts to Insert Polish-Lithuanian Cultural Heritage into Games". In *Proceedings of Digital Games Research Association - DiGRA 2023*, 1–16.

Ontañón, S., G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill and M. Preuss. 2013. "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft". *IEEE Transactions on Computational Intelligence and AI in Games* 5(4):293–311.

Poli, D., C. Berenotto, S. Blankenship, B. Piatkowski, G. A. Bader and M. Poore. 2012, 02. "Bringing Evolution to a Technological Generation: A Case Study with the Video Game SPORE". *The American Biology Teacher* 74(2):100–103.

Pozzer, C., J. Martins, L. Fontoura, L. Silva, M. Rutzig, R. Nunes *et al*. 2022. "SIS-ASTROS: An Integrated Simulation System for the Artillery Saturation Rocket System (ASTROS)". In *12th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2022)*, 194–201: SCITEPRESS – Science and Technology Publications, Lda.

Ricardo, D. C. 2021. "Friendly lords implementing an artificial intelligence social architecture in Mount & Blade II: Bannerlord". Master's thesis, Universidade do Algarve. https://sapientia.ualg.pt/entities/publication/284a70b1-8cfb-4b25-9edc-6469080d6a5d, accessed 15th August 2024.

Rogers, K. D. and A. A. Skabar. 2014. "A Micromanagement Task Allocation System for Real-Time Strategy Games". *IEEE Transactions on Computational Intelligence and AI in Games* 6(1):67–77.

Schrader, P. G., H. Deniz, and J. Keilty. 2016. "Breaking SPORE: Building Instructional Value in Science Education using a Commercial, Off-the Shelf Game". *Journal of Learning and Teaching in Digital Age* 1(1):63–73.

Synnaeve, Gabriel and Nardelli, Nantas and Auvolat, Alex and Chintala, Soumith and Lacroix, Timothée and Lin, Zeming and Richoux, Florian and Usunier, Nicolas 2016. "TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games". https://arxiv.org/abs/1611.00625, accessed 15th August 2024.

AUTHOR BIOGRAPHIES

EDISON PIGNATON DE FREITAS is Associate Professor on the Institute of Informatics at Federal University of Rio Grande do Sul, Brazil. As a reserve engineer officer of the Brazilian Army (rank of Captain), his research interests include military simulation systems, networked command and control systems, autonomous cooperative systems, computer networks, real-time systems, and unmanned aerial vehicles. His email address is: epfreitas@inf.ufrgs.br.

ELIAKIM ZACARIAS is a Computer Scientist and software developer in the SIS-ASTROS project of the Federal University of Santa Maria, Brazil. His research interests include artificial intelligence focused on pathfinding and autonomous behaviors, virtual learning software, and gamification software. His e-mail address is: eliakim.zacarias@gmail.com.

RAUL CERETTA NUNES is Full Professor at the Applied Computing Department in the Technological Center of Federal University of Santa Maria, Brazil. He has a Master's degree (1993) and a Doctor degree (2003) in Computer Science conferred by the Federal University of Rio Grande do Sul, Brazil. His research interests are distributed simulation, network security, and fault tolerance. His e-mail address is: ceretta@inf.ufsm.br.

LUIS ALVARO LIMA SILVA is Associate Professor at the Applied Computing Department in the Technological Center of Federal University of Santa Maria, Brazil. He has a PhD degree (2010) in Computer Science from University College London, UK. His research involves the advance of machine learning field for the development of agent-based simulation systems, health systems and computer games. His e-mail address is: luisalvaro@inf.ufsm.br.