applied sciences

MDPI

Article

MBSE Testbed for Rapid, Cost-Effective Prototyping and Evaluation of System Modeling Approaches

Azad M. Madni

Systems Architecting and Engineering Program, School of Engineering, University of Southern California, Los Angeles, CA 90089, USA; azad.madni@usc.edu

Featured Application: Perimeter security of stationary aircraft on a landing strip; autonomous vehicle navigation with dynamic obstacles.

Abstract: Model-based systems engineering (MBSE) has made significant strides in the last decade and is now beginning to increase coverage of the system life cycle and in the process generating many more digital artifacts. The MBSE community today recognizes the need for a flexible framework to efficiently organize, access, and manage MBSE artifacts; create and use digital twins for verification and validation; facilitate comparative evaluation of system models and algorithms; and assess system performance. This paper presents progress to date in developing a MBSE experimentation testbed that addresses these requirements. The current testbed comprises several components, including a scenario builder, a smart dashboard, a repository of system models and scenarios, connectors, optimization and learning algorithms, and simulation engines, all connected to a private cloud. The testbed has been successfully employed in developing an aircraft perimeter security system and an adaptive planning and decision-making system for autonomous vehicles. The testbed supports experimentation with simulated and physical sensors and with digital twins for verifying system behavior. A simulation-driven smart dashboard is used to visualize and conduct comparative evaluation of autonomous and human-in-the-loop control concepts and architectures. Key findings and lessons learned are presented along with a discussion of future directions.

Keywords: model-based systems engineering; MBSE; digital engineering; testbed; digital twin; simulation; ontologies

Citation: Madni, A.M. MBSE Testbed for Rapid, Cost-Effective Prototyping and Evaluation of System Modeling Approaches. *Appl. Sci.* **2021**, *11*, 2321. https://doi.org/10.3390/app11052321

Academic Editor: Dov Dori

Received: 16 January 2021
Accepted: 25 February 2021
Published: 5 March 2021

1. Introduction

Model-based systems engineering (MBSE) is making impressive strides both in increasing systems life-cycle coverage [1] and in the ability to model increasingly more complex systems [2,3]. Recently, MBSE has begun to employ the digital-twin concept [4] from digital engineering (DE) to enhance system verification and validation. Not surprisingly, these developments are increasingly producing more MBSE artifacts [5] that need to be organized, metadata-tagged, and managed to facilitate rapid development, integration, and "test drive" of system models in simulation in support of what-if experimentation.

Currently, MBSE researchers work with specific models and simulations to address a particular problem, thereby producing mostly point solutions. Furthermore, they seldom document assumptions and lessons learned. This practice implies that most MBSE researchers are largely starting without the benefit of the knowledge gained by others. Fortunately, MBSE is a methodology-neutral construct that allows researchers to pursue different modeling approaches, experiment with different algorithms, and learn from such experiences. Most recently, in an attempt to make modeling more rigorous, MBSE researchers are turning to formal ontologies to underpin their system models and facilitate assessment of model completeness, semantic consistency, syntactic correctness, and traceability. In light of these deficiencies and emerging trends and opportunities, this paper

introduces the concept of a MBSE testbed to organize and manage MBSE artifacts, support MBSE experimentation with different models, algorithms, and data, and catalogue the case studies and lessons learned. It presents a prototype implementation of the testbed and demonstrates the capabilities of the testbed for real-world operational scenarios, along with findings and lessons learned. It also presents an illustrative quantitative analysis of results produced through simulation-based experimentation.

2. Materials and Methods

The testbed encompasses the following: software programming environment; multiple modeling methods; analysis and optimization algorithms; repositories packages and libraries; simulation environments; and hardware components and connectors. The following list presents the core components of the testbed.

Software Programming Environment

- Operating system(s): Linux (Ubuntu 16.04) and Windows
- Programming environment: Spyder cross-platform development environment for Python 3.x programming language

Systems Modeling Methods (descriptive, analytic; deterministic, probabilistic)

- Systems Modeling Language (SysML) [6]
- Decision trees
- Hidden Markov Models (HMM) [7]
- Partially Observable Markov Decision Process (POMDP) model [8]

Optimization, Control, and Learning Algorithms

- Optimization using fitness functions
- N-step Look-ahead decision-processing algorithm
- Traditional deterministic control algorithm (e.g., proportional-integral-derivative controller (PID) algorithm)
- Q-learning algorithm

Repositories, Packages, Libraries

- NumPy—a Python library for manipulating large, multidimensional arrays and matrices
- Pandas—a Python library for data manipulation and analysis, specifically numerical tables, and time series
- Scikit-learn—a machine-learning library for Python; built on top of NumPy, SciPy and matplotlib
- Python-open CV—a library of Python bindings designed to solve computer vision problems

Simulation Platforms

- *Hardware–Software Integration Infrastructure:* Multiple simulation platforms are used for visualization, experimentation, and data collection from scenario simulations for both ground-based and airborne systems; integrate simulations (implemented in Python 3) with hardware platforms, such as Donkey Car [4] and quadcopters.
- *CARLA Simulation Platform:* CARLA offers an open-source high-fidelity simulator (including code and protocols) for autonomous driving research [5]. CARLA provides open digital assets, such as urban layouts, buildings, city maps, and vehicles. CARLA also supports flexible specification of sensor suites, environmental conditions, and dynamic actors. CARLA provides Python APIs to facilitate integration.
- *DroneKit Platform:* DroneKit, an open-source platform, is used to create apps, models, and algorithms that run on onboard computers installed on quadcopters. This platform provides various Python APIs that allows for experimenting with simulated quadcopters and drones. The code can be accessed on GitHub [9].

Hardware and Connectors

- Donkey Car platform: an open-source platform for conducting autonomous vehicles research
- Raspberry Pi (onboard computer): on the Donkey Car
- Quadcopters (very small UAVs used in surveillance missions), 1/16 scale robot cars
- Video Cameras: mounted on Donkey Car and quadcopters
- Socket communication: used to send commands from a computer to a Donkey Car or a quadcopter

3. Results

This section describes the research objectives, prototype implementation, and experimental results, along with key findings and implications.

3.1. Research Objectives

The key research objectives are to

- Develop a structured framework for cost-effective and rapid prototyping and experimentation with different models, algorithms, and operational scenarios.
- Develop an integrated hardware–software environment to support on-demand demonstrations and facilitate technology transition to customer environments.

The first objective encompasses the following:

1. Defining the key modeling formalisms that enable flexible modeling based on operational environment characteristics and knowledge of the system state space.
2. Defining a flexible and customizable user interface that enables scenario building by nonprogrammers, visualization of simulation execution from multiple perspectives, and tailored report generation.
3. Defining operational scenarios encompassing both nominal and extreme cases (i.e., edge cases) that challenge the capabilities of the system of interest (SoI)

The second objective encompasses the following:

4. Identifying low-cost components and connectors for realizing capabilities of the SoI
5. Defining an ontology-enabled integration capability to assure correctness of the integrated system.
6. Testing the integrated capability using an illustrative scenario of interest to the systems engineering community

These objectives are satisfied through the realization of a MBSE testbed specifically designed for rapid prototyping, what-if experimentation, data collection, and analysis.

3.2. MBSE Testbed Concept

The MBSE testbed concept is broader than that of conventional hardware-in-the-loop (HWIL) testbeds. HWIL testbeds are used for early integration and testing of physical systems as well as formal verification and validation (V&V). Typical HWIL testbeds consist of hardware, software modules, and simulations in which system components are either physically present or simulated. Simulated components are progressively replaced by physical components as they become available [10]. The MBSE testbed construct extends HWIL capabilities by including the means for developing and exercising abstract models independently or interoperating with HWIL [11–15]. Just as importantly, the MBSE testbed construct provides a modeling, simulation, and integration environment for developing and evaluating digital twins [5]. The envisioned capabilities of the MBSE testbed include the ability to:

- *represent models at multiple scales* and from different perspectives.
- *integrate with digital twins* of physical systems—support both symbolic and high fidelity, time-accurate simulations; the latter can be augmented by FPGA-based development

boards to create complex, time-critical simulations; and maintain synchronization between real-world hardware and virtual simulation.

- *accommodate multiple views*, multiple models, analysis tools, learning algorithms.
- *manage dynamically configurable systems* through software agents, employed in the simulation—in the future these agents could invoke processes allocated to Field Programmable Gate Arrays (FPGAs).
- *compose interoperable systems* with requisite flexibility to satisfy mission needs [10,16]
- *collect and generate evidence* for developing trust in systems.
- *exploit feedback* from the system and environment during adaptive system operation.
- *address temporal constraints* and their time-related considerations.
- *address change cascades* (e.g., arising from failures) not addressed by existing tools.
- *validate models*, which implies flexibility in simulation interfaces as well as propagation of changes in modeled components as data are collected in physical tests.

The testbed offers a means to improve understanding of functional requirements and operational behavior of the system in a simulated environment. It provides measurements from which quantitative characteristics of the system can be derived. It provides an implementation laboratory environment in which modeled real-world systems (i.e., digital twins) can be evaluated from different perspectives.

A testbed, in essence, comprises three components: an experimentation subsystem; a monitoring and measurement system; and a simulation–stimulation subsystem. The experimentation subsystem comprises real-world system components and prototypes which are the subject of experimentation. The monitoring and measurement system comprises interfaces to the experimentation system to extract raw data and a support component to collate and analyze the collected information. The simulation–stimulation subsystem provides the hooks and handles to experiment with real-world system agents and outputs to ensure a realistic experimentation environment.

However, testbeds can have limitations. For example, they cost too much, and they are limited to modeling systems and components that satisfy the testbed environment constraints. In addition, for some problems, analytic and/or simulation models may be more appropriate. This would be the case for complex distributed systems. Therefore, testbeds can be viewed as a flexible and modeler platform that complements/subsumes simulation and analytic methods.

Figure 1 presents the MBSE testbed concept. The testbed comprises a user interface that supports the following: scenario authoring, dashboard capabilities for scenario execution monitoring visualization, and control, and report generation; a suite of modeling and analysis tools including system modelers, machine learning and data analytics algorithms; simulation engines for discrete event simulation, hybrid simulation, and component simulation; and repositories of operational scenario vignettes, system models, component libraries, and experimentation results.

The testbed supports system conceptualization, realization, and assurance. *System conceptualization* comprises use case development; requirement elicitation, decomposition, and allocation; system concept of operations (CONOPS); logical architectures; metrics; initial system models; and initial validation concepts. *System realization* entails detailed design, physical and simulation development, and integration and test. *System assurance* comprises evaluating system safety, security, and mission assurance.

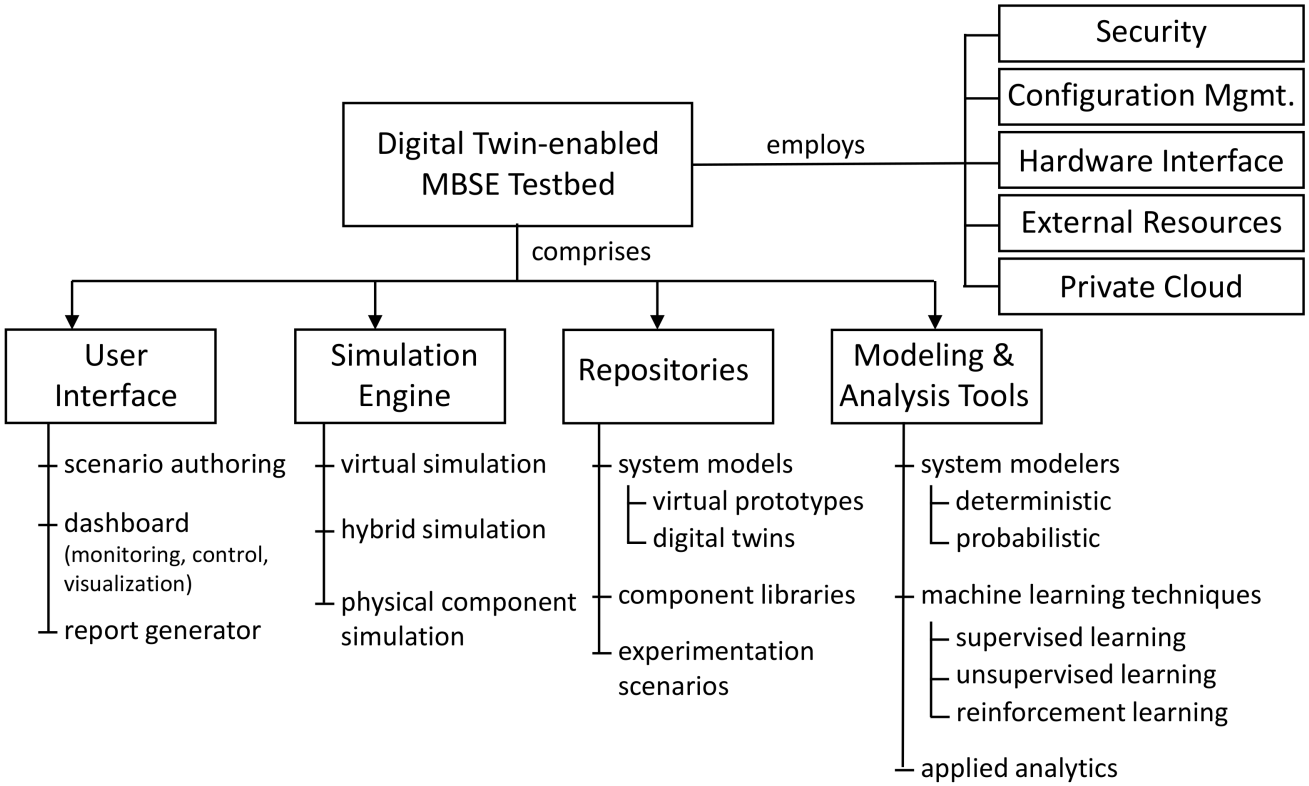


Figure 1. Model-based systems engineering (MBSE) testbed concept.

Complex systems are invariably a combination of third-party components and legacy components from previously deployed systems. As such, some components tend to be fully verified under certain operating conditions that may or may not apply in their reuse. Furthermore, complex systems are subject to unreliable interactions (e.g., sporadic or incorrect sensor inputs, and control commands that are not always precisely followed) because they interact frequently with the physical world. Finally, with increasing connectedness, they are increasingly susceptible to security threats. In light of the foregoing, the MBSE testbed needs to provide:

- *Inheritance evaluation*, in which legacy and third-party components are subjected to the usage and environmental conditions of the new system.
- *Probabilistic learning models*, which begin with incomplete system representations and progressively fill in details and gaps with incoming data from collection assets; the latter enable learning and filling in gaps in the knowledge of system and environment states.
- *Networked control*, which requires reliable execution and communication that enables satisfaction of hard time deadlines [5,17] across a network. Because networked control is susceptible to multiple points of cyber vulnerabilities, the testbed infrastructure should incorporate cybersecurity and cyber-resilience.
- *Enforceable properties* define core attributes of a system that must remain immutable in the presence of dynamic and potentially unpredictable environments. The testbed must support verification that these properties are dependable regardless of external conditions and changes.
- *Commercial off-the-shelf (COTS) components*, which typically communicate with each other across multiple networks and time scales [5,18]. The latter requires validation of interoperability among COTS systems.
- *Support for safety-critical* systems in the form of, for example, executable, real-time system models that detect safety problems and then shut down the simulation, while the testbed can be queried to determine what happened.

3.3. Logical Architecture of MBSE Testbed

Incorporating the capabilities described in Section 3 into the testbed is being performed in stages. Figure 2 presents the initial logical configuration of the testbed.

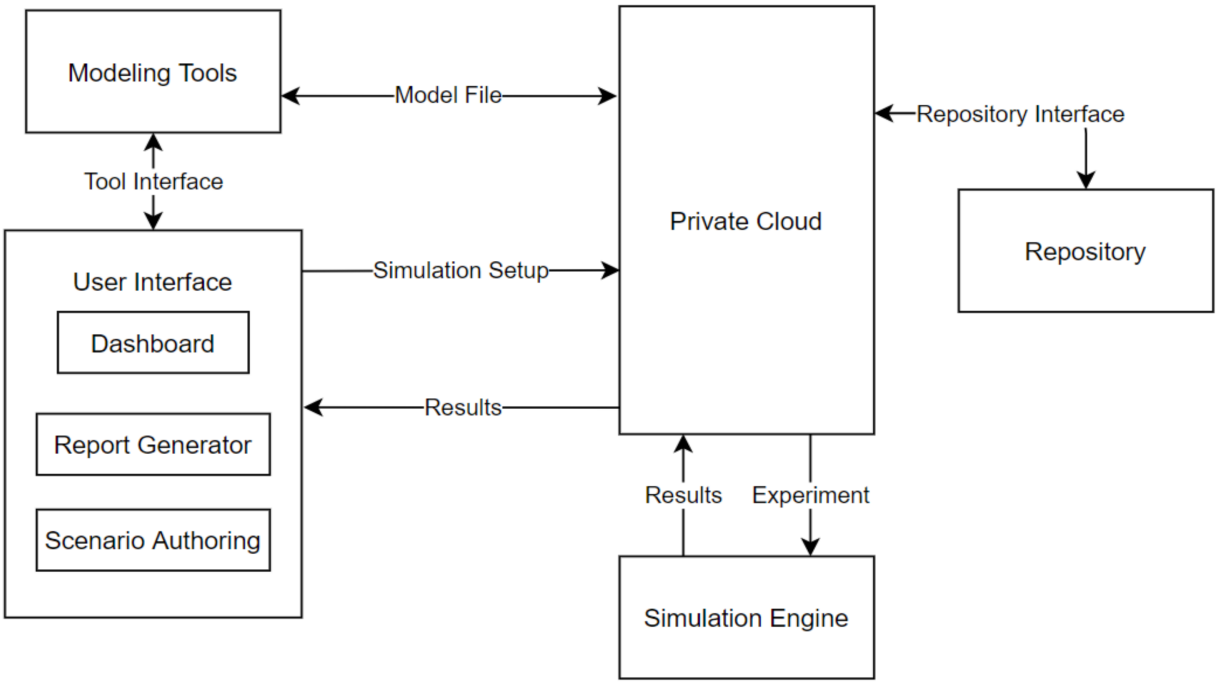


Figure 2. Prototype MBSE testbed logical architecture.

As shown in Figure 2, the testbed prototype comprises: (a) a user interface for scenario definition and system modeling as well as for the dashboard used for monitoring, visualization, and controlling scenario execution; (b) models created by the systems engineer or researcher that reflect an envisioned or existing system are stored in the repository; (c) a multiscenario capable simulation engine that dynamically responds to injects from the user interface and collects experiment results that area sent to the repository and user interface; (d) experiment scenarios stored in the repository or entered from the GUI; and (e) a private cloud that provides testbed connectivity and protects MBSE assets.

The prototype testbed implementation supports virtual, physical, and hybrid simulations. It supports virtual system modeling and interoperability with the physical system. It is able to access data (initially manually and eventually autonomously) from the physical system to update the virtual system model thereby making it into a digital twin of the physical system. The testbed supports proper switching from the physical system to the digital twin and vice versa using the same control software.

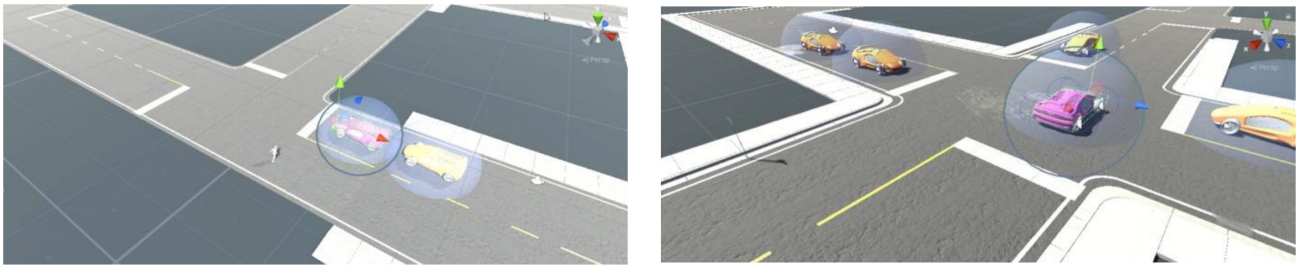
The prototype testbed currently offers the following capabilities.

3.3.1. System Modeling and Verification

The testbed offers both deterministic-modeling and probabilistic-modeling capabilities. In particular, it offers SysML modeling for deterministic systems and partially observable Markov decision process (POMDP) modeling for probabilistic systems. Exemplar models of both types are provided in an online "starter kit" to allow users to make a copy before commencing the system modeling activity. Verification in this context pertains to ascertaining model correctness (i.e., model completeness with request to questions that need to be answered, semantic and syntactic consistency, and model traceability to requirements).

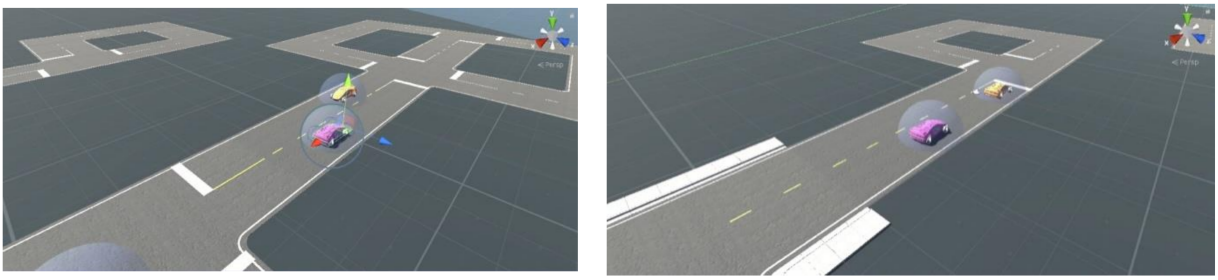
3.3.2. Rapid Scenario Authoring

Eclipse Papyrus is used along with SysML and Unity 3D virtual environment for scenario authoring and definition of entity behaviors. The testbed offers scenario authoring and visualization for multiple domains. For example, Figures 3 and 4 show the results of autonomous vehicle scenario authoring. Figure 5 shows visualizations for aircraft perimeter security scenario. These exemplar scenarios are used in experimenting with planning and decision-making models and algorithms. The initial scenario contexts are defined in SysML (Figure 6), with Python XMI being used to extract data from the SysML model to populate the Unity virtual environment.



(**a**) (**b**)

Figure 3. (**a**) Pedestrian crossing scenario, (**b**) four-way stop sign scenario.

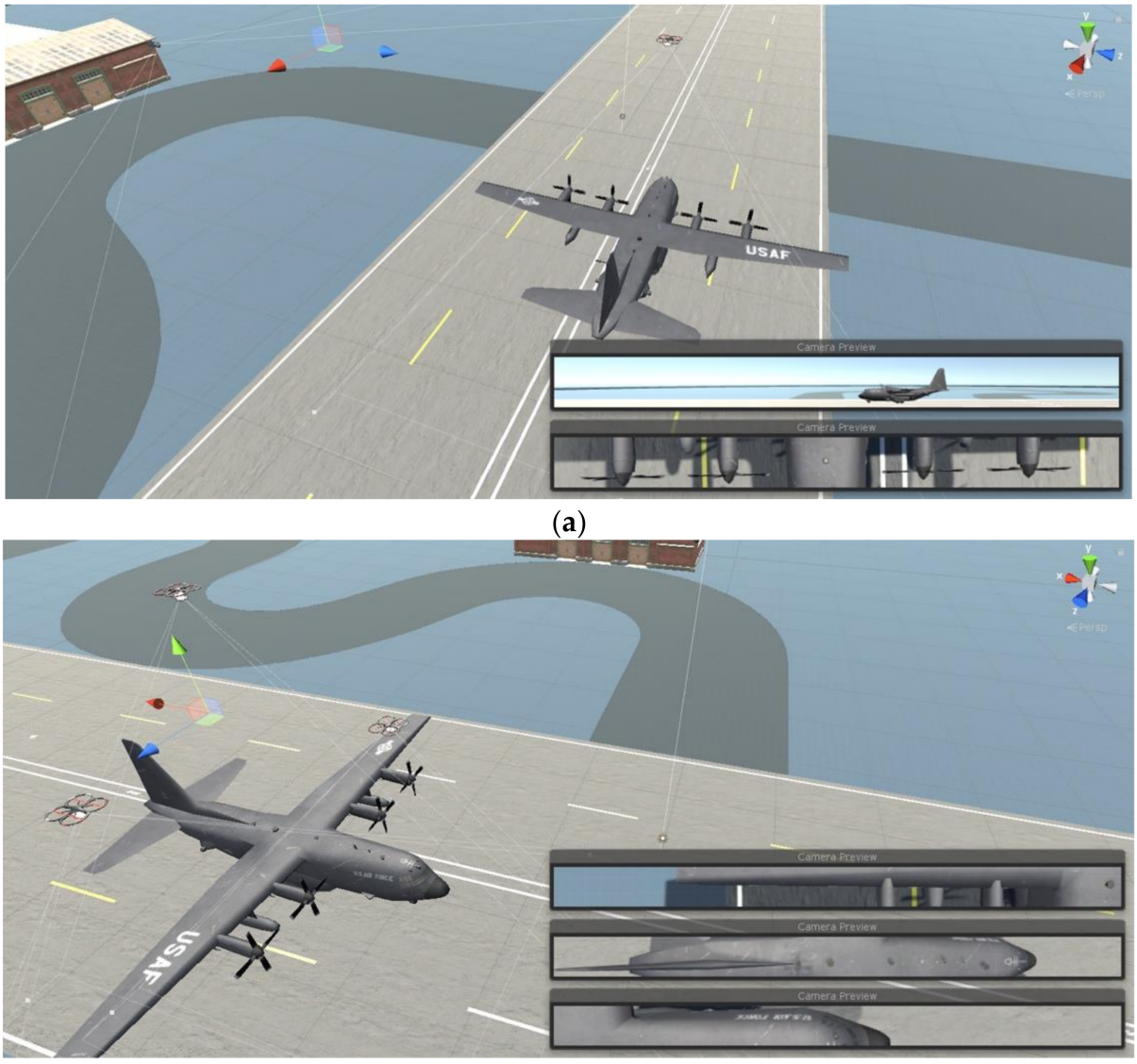


(**a**) (**b**)

Figure 4. (**a**) Vehicle crossing scenario, (**b**) vehicle braking scenario.

The behaviors of scenario entities such as autonomous vehicles, pedestrians, UAVs, and surveillance cameras are defined in Unity. The UAV, pedestrian, and vehicle behaviors are defined using standard waypoint-following algorithms in Unity. The planning and decision-making algorithms are exercised and tested with both autonomous vehicle navigation and control operations, and multi-UAV operations in aircraft perimeter security mission. Figure 3a presents a visualization of a pedestrian crossing scenario, while Figure 3b presents a visualization of a four-way stop sign scenario. Similarly, Figure 4a presents a visualization of a vehicle crossing scenario, while Figure 4b presents a visualization of a vehicle braking scenario.

Figure 5a depicts the aircraft perimeter security scenario with one UAV and one video camera conducting surveillance. On the bottom right corner of the figure, camera views are shown. Figure 5b presents the aircraft perimeter security with three UAVs. Eclipse Papyrus for SysML, Python XMI, and Unity 3D are used to rapidly author the scenarios with various agents. The scenarios have static agents, perception agents, dynamic auxiliary agents, and system-of-interest (SoI) agents. Static agents such as standing aircraft, traffic signs, and buildings are part of the scenario context. Perception agents, such as cameras that capture simulation environment data, are used for processing. Dynamic auxiliary agents, such as pedestrians and auxiliary cars, as well as auxiliary UAVs, follow predefined behavior in experiments. The system-of-interest (SoI) agents such as the autonomous car or UAV are used to test different algorithms defined by the experimenters.



(**a**)

(**b**)

Figure 5. (**a**) Scenario with UAV and surveillance cameras, (**b**) scenario with multiple surveillance UAVs.

Figure 6 presents a SysML representation of scenario context for the aircraft perimeter security example. Included in the context are the UAV, airstrip, building, surveillance camera, and aircraft. Attributes and operations of each of those entities are defined in specific blocks.

3.3.3. Model and Scenario Refinement

The experiment/test scenarios allow rapid and straightforward substitution of refined models for coarse models. In addition, hardware components can be substituted for virtual models. In Unity 3D, it is possible to extract various relevant properties of scenario objects such as velocities, locations, and states. Entity behaviors are assigned to objects using Unity 3D scripts written in C#. This capability affords greater flexibility in experimentation. A Python interface is used for testing various machine-learning (ML) algorithms.

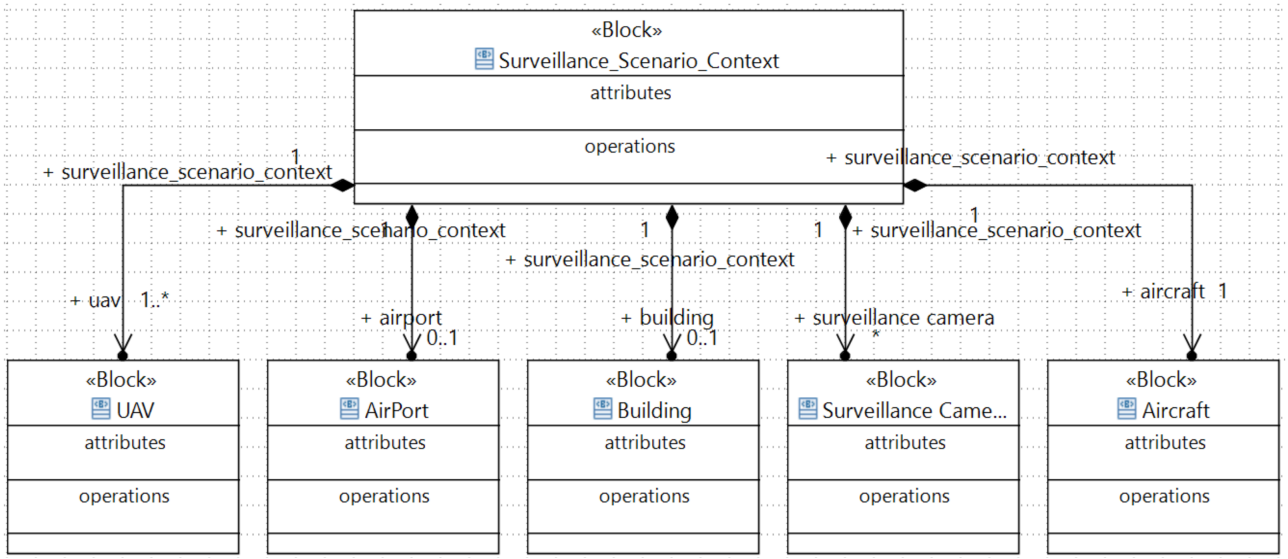


Figure 6. Scenario context definition.

3.3.4. MBSE Repository

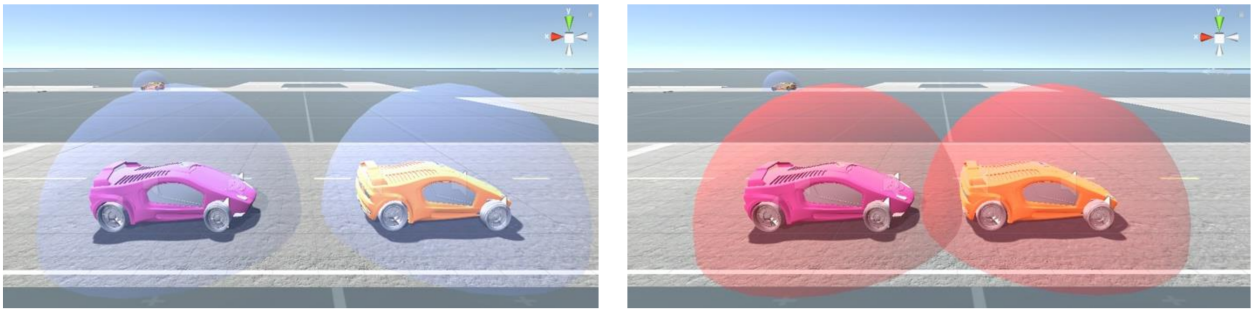
The testbed repository contains libraries of scenarios, scenario objects, 3D objects, behaviors, and system/component models. For example, 3D objects such as UAV models, ground vehicle models, pedestrian models, and camera models are part of the scenario object repository. The model repository comprises system models and behaviors that can be associated with various objects in the scenario.

3.3.5. Experimentation Support

The testbed's virtual environment supports the collection and storage of data from experiments. For example, variables such as distances between vehicles, velocities, and decisions made by autonomous vehicles in various scenarios can be captured and stored for post hoc analysis. Data collected during experimentation can be used by machine-learning algorithms to train models. The MBSE testbed provides access to the properties of scenario objects. For example, velocity, size, shape, and location of static objects and auxiliary agents are directly extracted from the virtual environment. This capability enables the creation of feedback loops and facilitates the definition of abstract perception systems of autonomous vehicles or SoI agents. C# scripts are used to extract, process, and transfer data to other components of the dashboard. The virtual environment allows for manual control of objects, thereby affording additional flexibility in experimentation and testing. Multiple human users are able to interact with virtual objects, thereby realizing complex behaviors during experimentation.

3.3.6. Multiperspective Visualization

The virtual environment offers visualization of system behaviors during simulation in intuitive ways. Exemplar visualizations for self-driving cars and multi-UAV operations are presented in Figures 7–9. Figure 7a,b, show the change in the state of the car from "safe-state" (blue clouds surrounding the cars) to "about-to-collide state" (red clouds surrounding the cars). Figure 8 shows another visualization perspective of the car trajectory. The purple car is the system of interest. The red zones along the vehicle trajectory indicate constraints.



(**a**) (**b**)

Figure 7. (**a**) Cars with "safe states", and (**b**) cars with about-to-collide state.

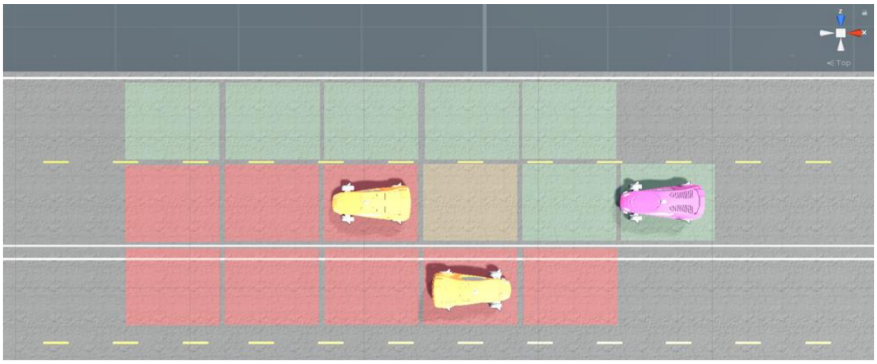
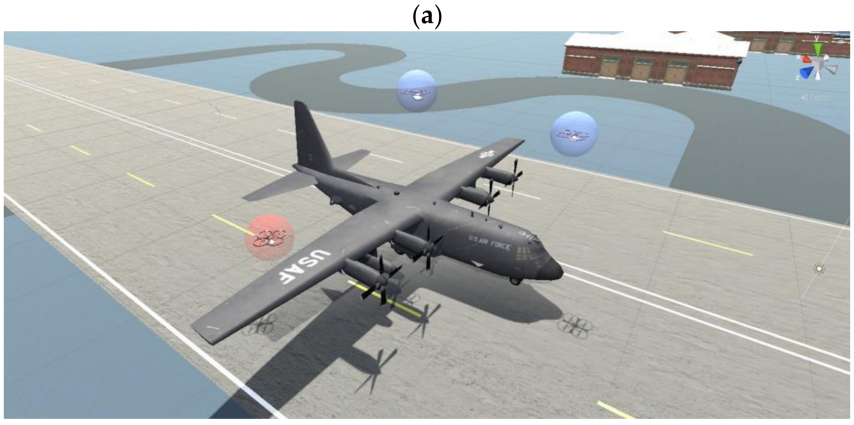


Figure 8. Visualize car trajectory.

(**a**)



(**b**)

Figure 9. (**a**) All UAVs in "safe states", and (**b**) one UAV in "unsafe state".

Figure 9 presents examples of visualizations for multi-UAV operations. Figure 9a,b show the change in the state of the UAV from "safe-state" (blue cloud surrounding the UAV) to "unsafe-state" (red cloud surrounding the UAV). Virtual environments offer a convenient means for stakeholders to contribute to what-if experimentation. Figure 10 shows another perspective in which UAV trajectories can be visualized during experimentation with planning and decision-making algorithms. These visualization assets and respective scripts are stored in the repository. The experimenter can drag and drop the asset on the scenario object and integrate it with the experiment. The assets have a user interface to customize the visualization parameters.

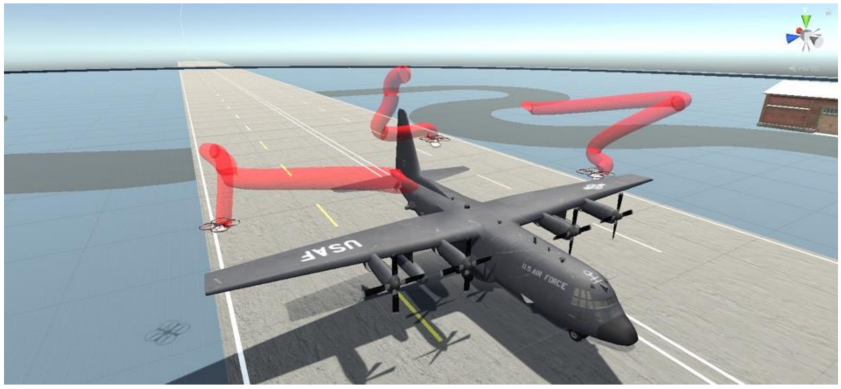


Figure 10. UAV trajectory visualization.

3.4. Implementation Architecture

Figure 11 provides the implementation architecture of the testbed. As shown in the figure, a private cloud interfaces with virtual simulation, physical component simulation, modeling tool, ontology representation, user interface, and analysis tool.

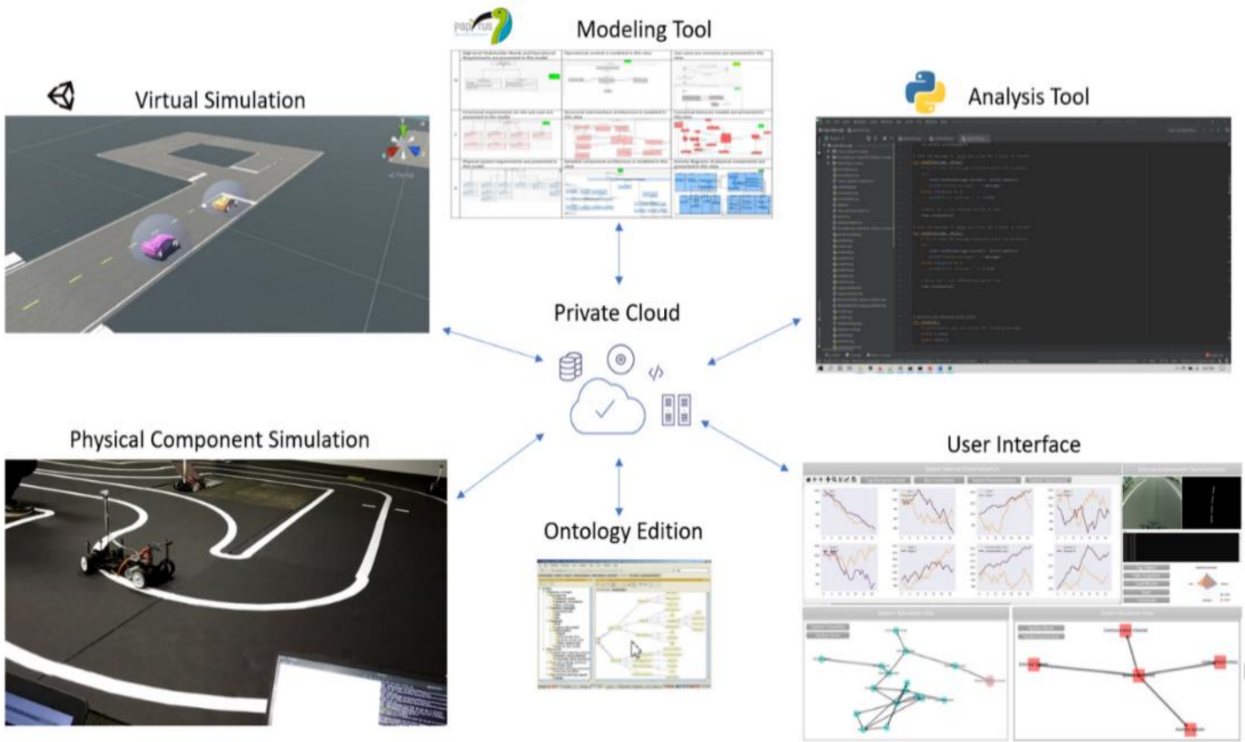


Figure 11. Testbed implementation architecture.

A ground-vehicle obstacle-avoidance scenario comprising multiple models was developed to demonstrate the utility and use of the testbed. This relatively simple scenario is used to demonstrate the integration of testbed components needed for model validation.

In this simple scenario, an autonomous vehicle has to drive safely behind another vehicle (which is viewed as a potential obstacle from a modeling perspective). The obstacle-avoidance scenario is represented by a state machine diagram in SysML (Figure 12). In this example, we define maintaining a distance of three meters as the safe distance between the two vehicles. Figure 12 shows that no action is taken when the vehicles are at least three meters apart. A transition occurs to the ActionState when the gap is less than three meters.

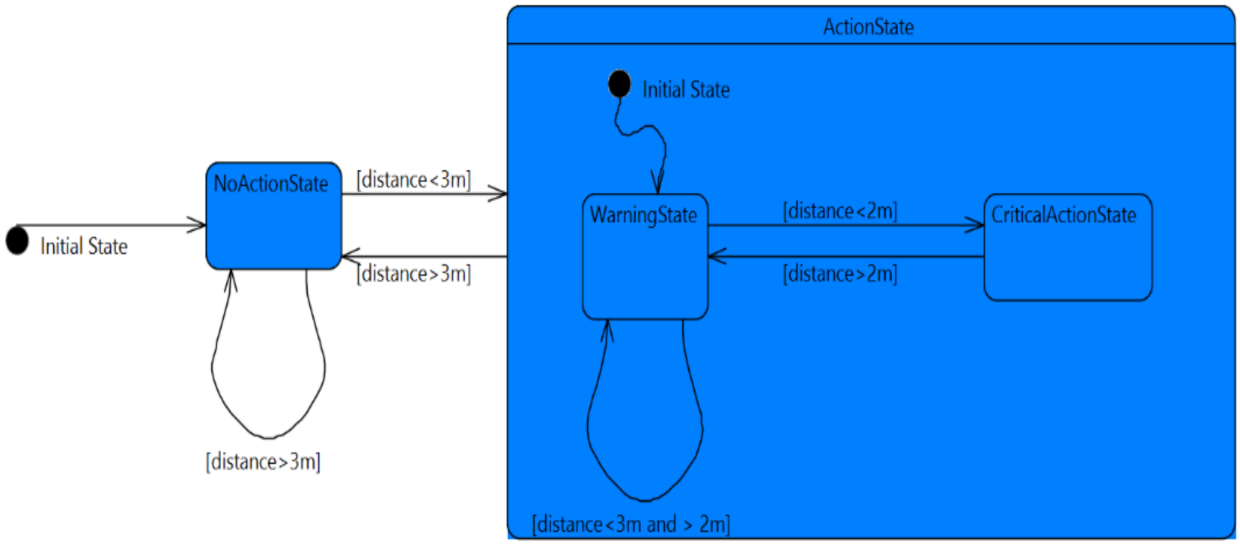


Figure 12. SysML high-level-state machine diagram for obstacle avoidance.

The SysML model is mapped to the 3D virtual environment. For this mapping, a Python XMI translation tool automatically populates the asset container in Unity 3D from SysML models. Objects in Unity 3D are stored in an asset container. Figure 13 presents the architecture of the SysML to 3D virtual environment translation. Figure 13 displays the SysML block definition diagram for the obstacle-avoidance scenario context. This diagram presents various entities that are part of the scenario. The blocks in the diagram are arranged under the "ScenarioEntities" package. The structured packaging of SysML model entities facilitates the extraction of model elements.

As shown in Figure 13, car, road, and obstacle-car blocks are extracted from the scenario context SysML model. These block names are then matched with existing 3D objects in the repository. When a match is found, the object is duplicated from the repository to the asset container of the 3D virtual environment. When a matching 3D object is not present in the repository, the translation program creates an empty object in the asset container. For example, in Figure 13, because the "obstacle car" car object is not present in the 3D repository, an empty object is created in the asset repository. Users can further model the 3D object as per requirement. Here, the car object is duplicated to create the obstacle-car object. The user can employ the asset container to populate objects in the Unity 3D simulation. In case of a change to the SysML model, a refresh function triggers the translation program to update the asset container. A user can further customize 3D objects in the virtual environment and then populate additional objects and behaviors from the repository.

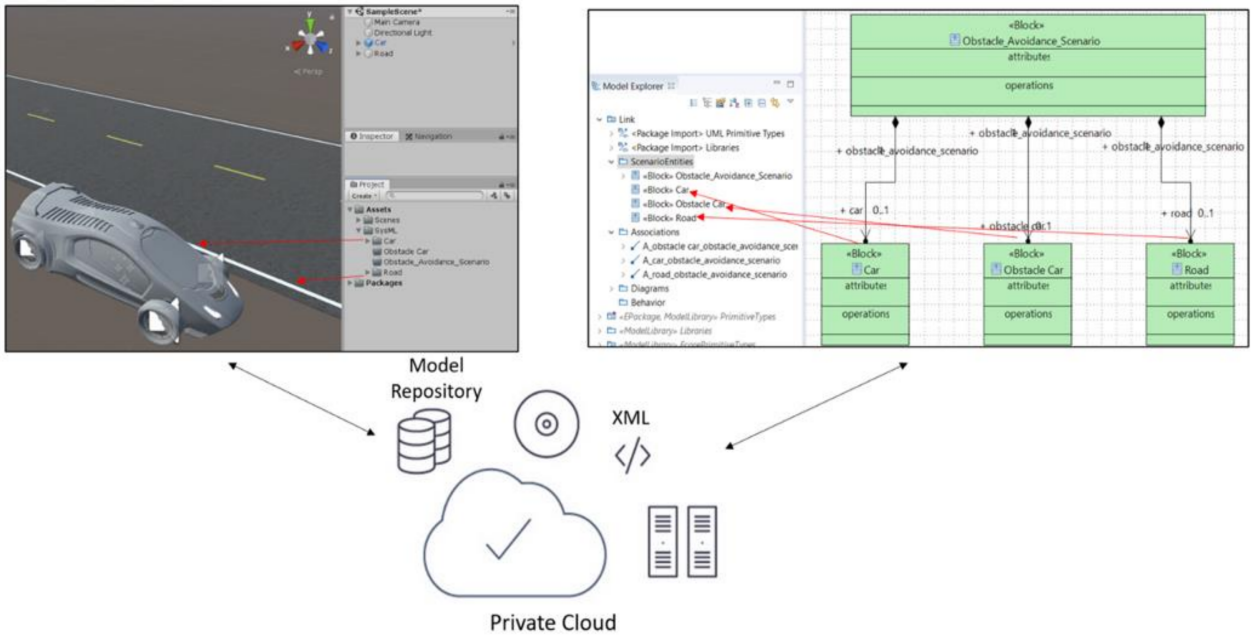


Figure 13. SysML to 3D virtual simulation mapping.

Waypoint-following behavior is assigned to the front vehicle in Unity 3D. A set of points and respective velocities to be followed by front vehicle are then assigned. The simulation generates navigation areas and trajectories for a given safe distance.

During the 3D simulation, we discovered that safe distances vary based on the velocities of the vehicles, which can impact the navigation area and trajectories. We conducted experiments that varied safe-distance values and vehicle velocities to assess the impact on navigation area and possible trajectories of the autonomous vehicle. Initially, for small values of safe distance, the autonomous vehicle has multiple paths to navigate around an obstacle, but as the safe distance between vehicles is increased, the safe navigation area around the vehicle shrinks. Figure 14 shows simulation results of changing the safe distance on navigation area around the obstacle. As expected, the simulation showed that the navigation area for the autonomous car shrinks when the front vehicle moves relatively slower. The simulation also uncovered an assumption error in our initial experiment resulting from the use of a fixed safe distance regardless of the vehicle's relative position and velocity.

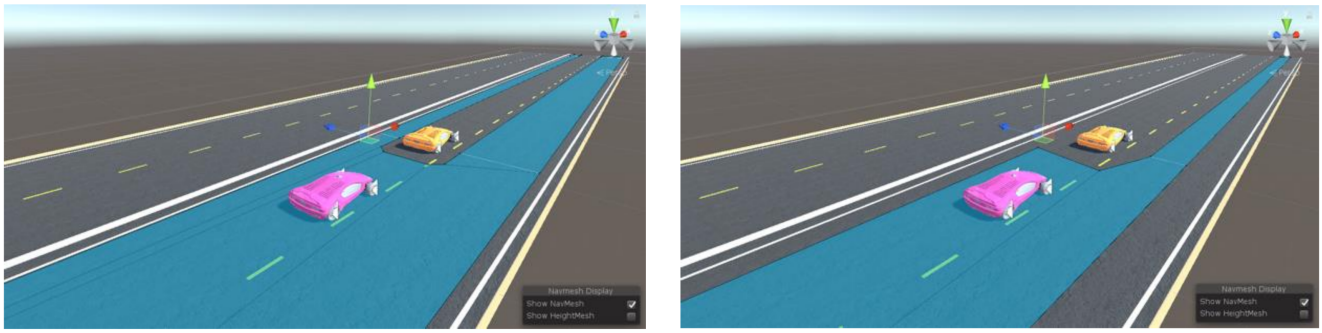


Figure 14. Simulation results of changing the safe distance on navigation area around the obstacle.

Currently, various testbed components can be integrated to explore and experiment with "what if" scenarios. In addition, conceptual models can be created and refined as needed. In the above simple experiment, it became evident that the overall model needed refinement to explicate variables such as longitudinal and lateral safe distances, vehicle velocities, and vehicle acceleration and braking capacities. The simulation confirmed

the obvious need for implicitly defined safe-distance rules, while also confirming that acceptable driving practices are context dependent. The ability to experiment with heterogeneous models and collect and analyze data to uncover patterns and trends enables more comprehensive elicitation of requirements. Repositories of 3D assets and their behaviors were used for rapid authoring of scenarios. For example, for the obstacle car that had a waypoint-following behavior in 3D simulation, the model of a car, road, and waypoints in the 3D environment were used from repositories and customized for a particular scenario. Additionally, algorithm visualization methods and behaviors available in the testbed repository were employed.

4. Quantitative Analysis

The MBSE testbed allows users to integrate and evaluate different machine-learning models with different parameters. We created an exemplar simulation environment. The exemplar simulation environment consists of a UAV in an indoor building setup searching for a predefined object. In the search mission, the UAV agent receives a negative reward for colliding with objects in the environment and a positive reward for touching the predefined goal object. Users can test multiple reinforcement learning algorithms on the UAV agent. We tested the integrated models with the UAV making observations in the environment and then taking corresponding actions. Five reinforcement learning algorithms were evaluated in the experiment: proximal policy optimization (PPO), soft actor-critic (SAC), PPO with generative adversarial imitation learning (GAIL), PPO with behavioral cloning (BC), and PPO combined with GAIL and BC [1–4]. Model parameters such as cumulative reward, episode length, policy loss, and entropy were evaluated against the number of simulation runs for selected reinforcement learning algorithms. Figure 15 presents the learning cycle, while Figure 16 presents the testbed setup for quantitative analysis.

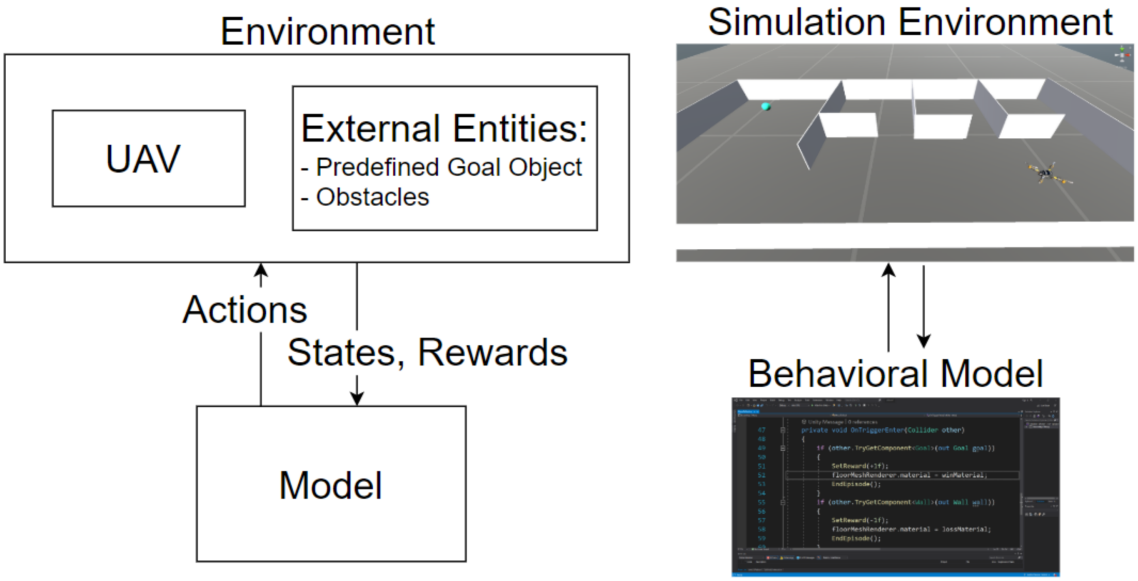


Figure 15. Learning cycle.

As shown in Figure 16, the structural model in SysML is mapped to the 3D virtual environment. For this mapping, a Python XMI translation tool was built. This tool automatically populates the asset container in the simulation environment using the SysML models. Objects in the simulation environment are stored in an asset container. Figure 16 displays the SysML block definition diagram for the "indoor search" scenario context. This diagram presents various entities that are part of the scenario. The blocks in the diagram are arranged under the "ScenarioEntities" package. The structured packaging of SysML model entities facilitates the extraction of model elements.

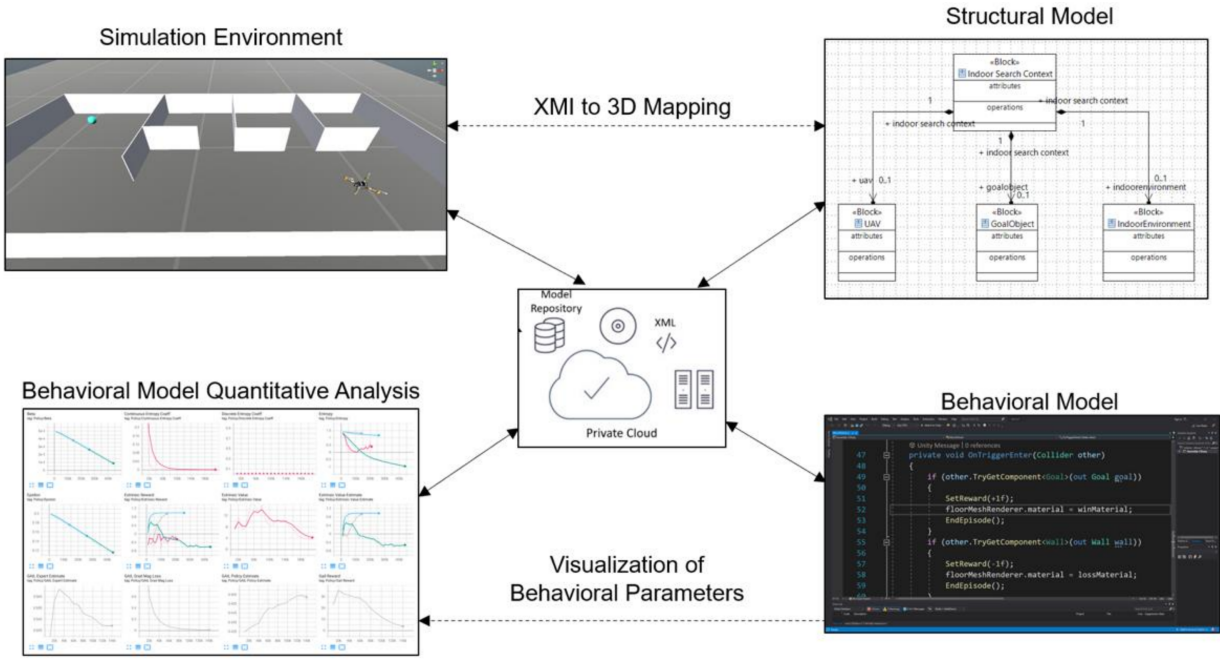


Figure 16. Quantitative analysis setup using testbed.

The UAV, indoor environment, and goal object blocks are extracted from the scenario context SysML model. These block names are then matched with existing 3D objects in the repository. When a match is found, the object in the repository is duplicated and inserted into the 3D virtual environment's asset container. When a matching 3D object is not present in the repository, the translation program creates an empty object in the asset container for the user to further modify. The user can then employ the asset container to populate objects in the simulation. Observations and actions are defined for the simulation setup, and the rewards mechanism is created in the experiment.

As noted earlier, five models were evaluated in the experiment: proximal policy optimization (PPO), soft actor-critic (SAC), PPO with generative adversarial imitation learning (GAIL), PPO with behavioral cloning (BC), and PPO combined with GAIL and BC [19–22]. We used the Unity ML agents kit to extend the testbed capabilities [23].

Figure 17 presents simulation runs on the horizontal axis, and cumulative rewards gained by the agent for a given model on the vertical axis. For PPO and PPO with GAIL, the mean cumulative episode reward increases as the training progress. For the rest of the agents, reward decreases, indicating that the agent is not learning successfully for the given simulation environment and for the given simulation runs.

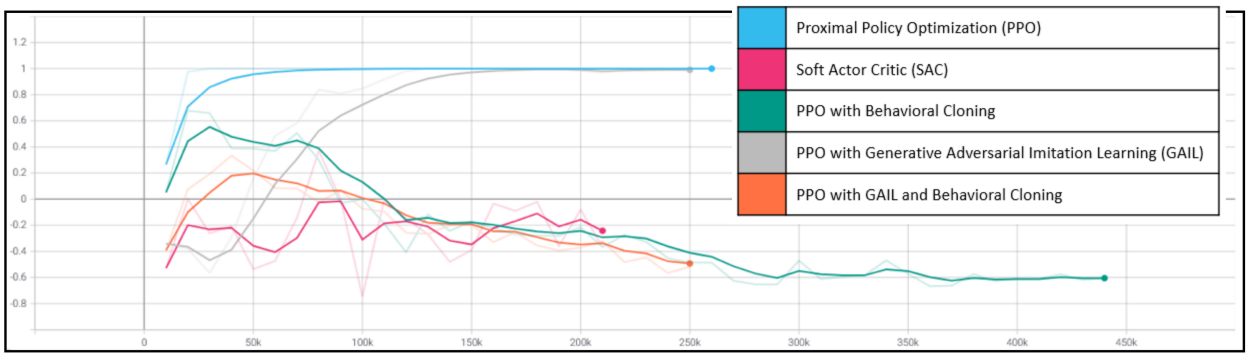
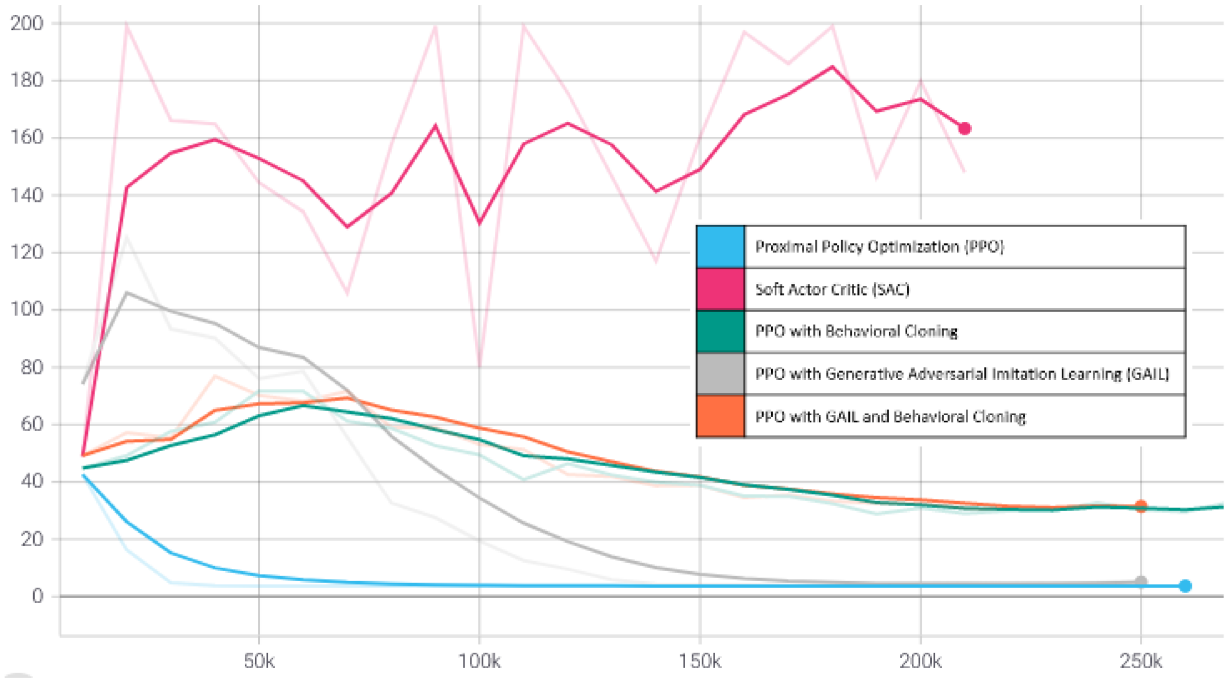
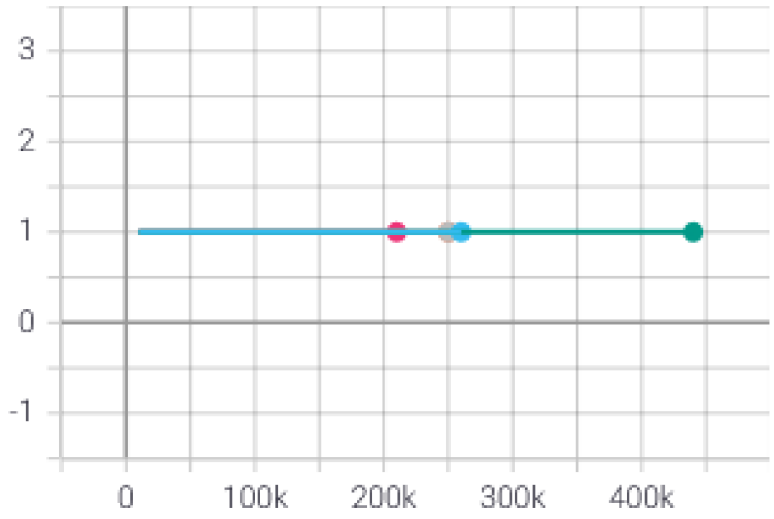


Figure 17. Simulation runs vs. cumulative reward.

Figure 18a shows that the mean length of episodes goes down in the environment for successful agents as the training progresses. The "is training" Boolean in Figure 18b indicates whether the agent is updating its model or not.



(**a**)



(**b**)

Figure 18. (**a**) Simulation runs vs. episode length, and (**b**) is training (Boolean).

The different models exhibit different behaviors for a given simulation set up. The way the user sets up the training environment impacts the performance of the models differently.

In Figure 19a,b, the policy loss parameter indicates how much the policy is changing for each agent. Various models have different profiles, and for most models, the magnitude of policy loss decreases indicating successful training. In Figure 20, the entropy measure indicates the degree of randomness of decisions made by the model. The slow decrease of this parameter is an indicator of a successful training session. Entropy profiles are different for different agents.

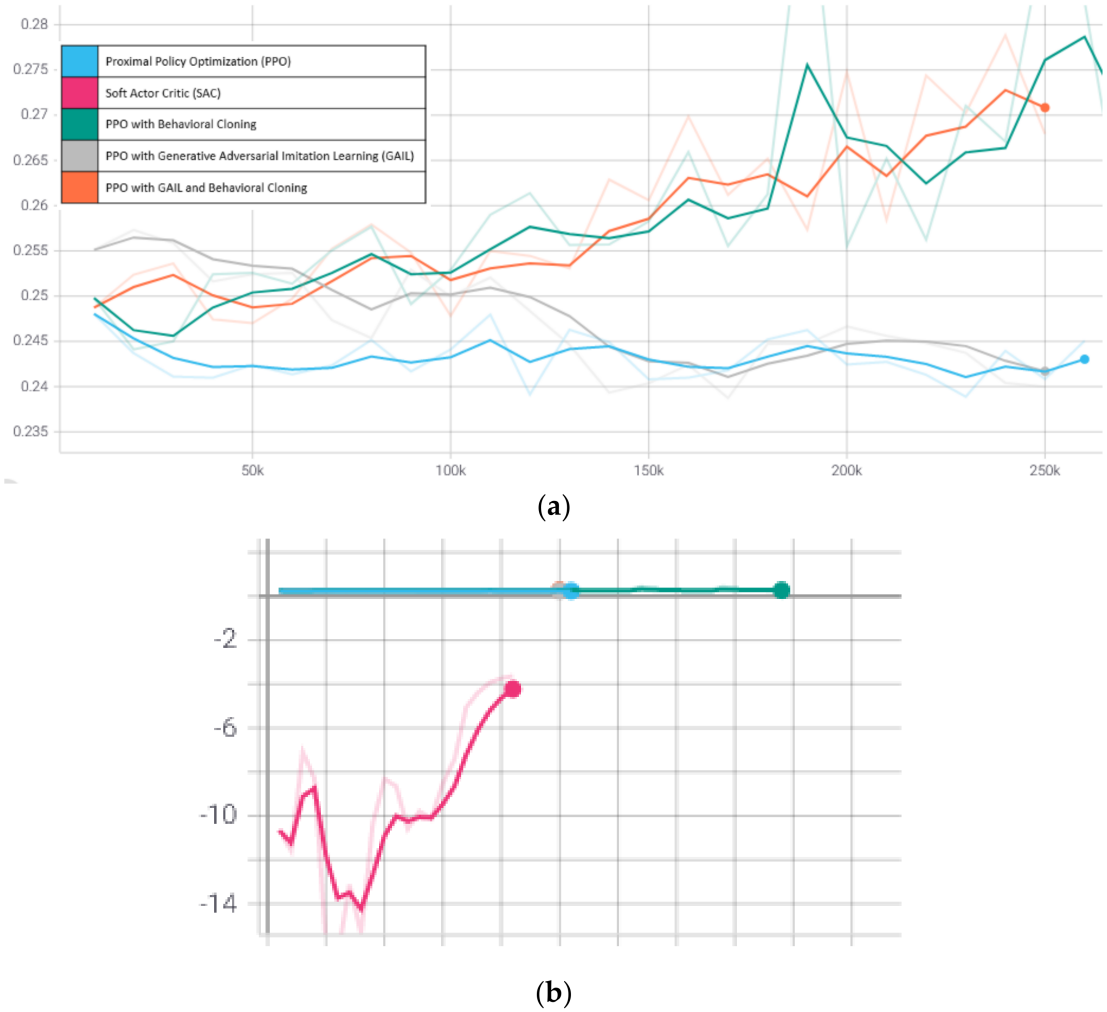


Figure 19. (**a**) Simulation runs vs. policy loss (excluding SAC), and (**b**) simulation runs vs. policy loss for all models.

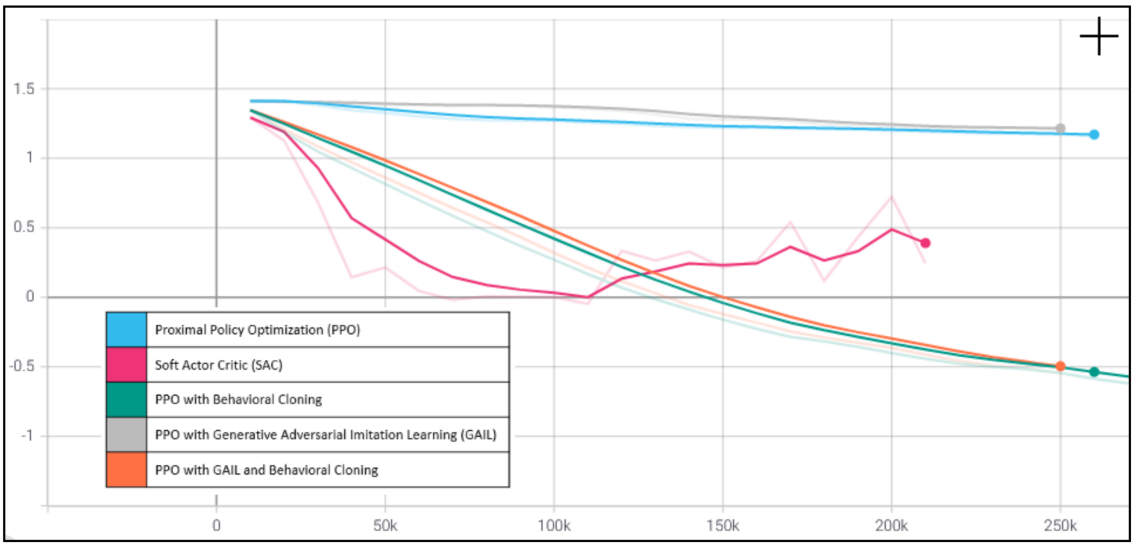


Figure 20. Simulation runs vs. entropy.

Additionally, it is possible to analyze specific model parameters such as GAIL expert estimate, GAIL policy estimate, and GAIL rewards (Figure 21).

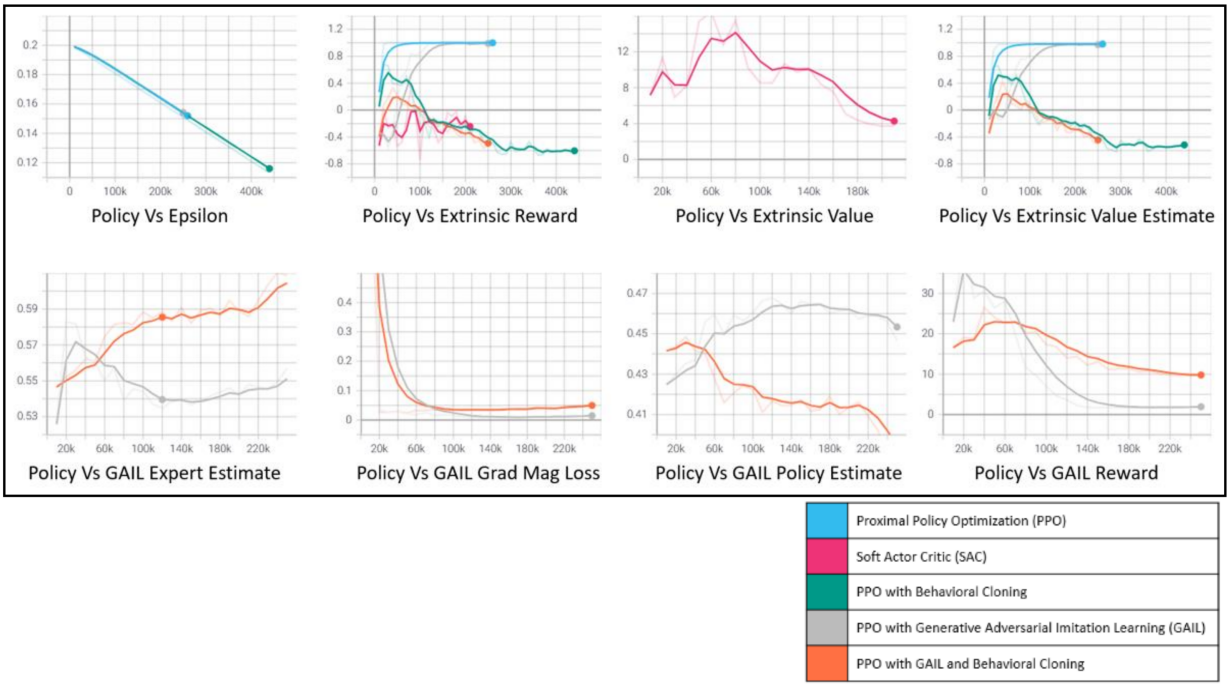


Figure 21. Model-specific parameter evaluation.

The testbed also allows quantitative analysis of behavioral models. Users can manipulate various model parameters and run experiments. The testbed capabilities enable users to formulate the decision, observation, and reward problem more holistically. Various use cases and scenarios can be considered before defining agent behaviors. The holistic approach and quantitative analysis allow users to determine effective strategies for intelligent agents.

5. Discussion

This paper has presented the system concept, architecture, prototype implementation, and quantitative analysis supported by a MBSE testbed that enables experimentation with different models, algorithms, and operational scenarios. Several important lessons were learned from the prototype implementation. First, a minimal testbed ontology [24] with essential capabilities can be quite useful to begin initial experimentation with models and algorithms. Second, it is possible to adapt system model complexity to scenario complexity and thereby minimize computation load when possible. For example, for simple driving scenarios, a finite state machine (FSM) can suffice for vehicle navigation. However, as the driving scenario gets more complicated (e.g., poor observability and uncertainties in the environment), more complex system models such as POMDP can be employed to cope with environment uncertainty and partial observability. The value of POMDP modeling becomes evident when operating in complex uncertain environments. Third, when it comes to system development, it pays to start off with simple scenarios and ensure that safety requirements are met, and then progressively complicate driving scenarios and employ more complex models while continuing to assure satisfaction of safety requirements. The most straightforward way to implement this strategy is to control the complexity of the operational environment by imposing constraints (e.g., have a dedicated lane for autonomous vehicles or avoid driving in crowded streets). Then, after the simple model has been shown to satisfy safety requirements, constraints can be systematically relaxed to create more complex driving scenarios. In the latter case, more complex system models can be employed, verified, and validated with respect to safety requirements. Fourth, system and environment models can be reused, thereby reducing development time. To facilitate model reuse, the models can be metadata-tagged with usage context. Then, contextual similarity between a problem situation and system models can be employed to determine

suitability of a particular system model for reuse in a particular context. This reuse feature can accelerate experimentation and development. Fifth, a smart, context-sensitive, scenario-driven dashboard can be used to dynamically adapt monitoring capability and dashboard display to maximize situation awareness with manageable cognitive load. To this end, the familiar METT-TC construct employed by the military can be employed as the underlying ontology. Based on the prevailing context, either the total set or a subset of variables in METT-TC may be applicable to characterize context. The flexible architecture of the scenario-driven dashboard can support both human-in-the-loop and autonomous operations. Just as importantly, it provides a convenient and cost-effective environment to try out different augmented-intelligence concepts [25]. Sixth, a hybrid, distributed simulation capability enables integration of virtual simulation with real-world systems (e.g., self-driving cars and unmanned aerial vehicles). This integration, in turn, enables the creation of digital twins which can enhance system verification and validation activities [5]. Furthermore, by assuring individual control of simulations, the testbed offers requisite flexibility in experimentation with system/system-of-system simulation. In addition, by allowing different models (e.g., scenario model and threat model) to run on different computers, simulation performance can be significantly increased. Finally, an illustrative quantitative analysis capability is presented to convey how simulation results can be analyzed to generate new insights.

Future directions include the creation of formal ontologies and metamodel [24] to guide systems integration, human–systems integration [26,27], adversarial modeling, introduction of digital twins at the system, and subsystem levels [5], reinforcement learning techniques to cope with partial observability and uncertainty [17], support for distributed simulation standards (i.e., IEEE 1278.2-2015), and ontology-enabled reuse [28,29] and interoperability [30].

Funding: This research was funded, in part, by the US Department of Defense Systems Engineering Research Center (SERC), agreement number: 2103056-01, under prime contractor number: HQ003419D0003.

Institutional Review Board Statement: Not Applicable.

Informed Consent Statement: Not Applicable.

Data Availability Statement: Not Applicable.

Acknowledgments: The author acknowledges the technical support provided by Ayesha Madni, Shatad Purohit of the University of Southern California, and Carla Madni of Intelligent Systems Technology, Inc.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Madni, A.M.; Sievers, M. Model-Based Systems Engineering: Motivation, Current Status, and Research Opportunities. *Syst. Eng.* **2018**, *21*, 172–190. [CrossRef]
2. Madni, A.M.; Purohit, S. Economic Analysis of Model Based Systems Engineering. *Systems* **2019**, *7*, 12.
3. Purohit, S.; Madni, A.M. Towards Making the Business Case for MBSE. In Proceedings of the 2020 Conference on Systems Engineering Research, Redondo Beach, CA, USA, 8–10 October 2020.
4. Schluse, M.; Atorf, L.; Rossmann, J. Experimental Digital Twins for Model-Based Systems Engineering and Simulation-Based Development. In Proceedings of the 2017 Annual IEEE International Systems Conference, Montreal, CA, USA, 24–27 April 2017; pp. 1–8.
5. Madni, A.M.; Madni, C.C.; Lucero, D.S. Leveraging Digital Twin Technology in Model-Based Systems Engineering. *Systems* **2019**, *7*, 7. [CrossRef]
6. Friedenthal, S.; Moore, A.; Steiner, R. *A Practical Guide to SysML: The Systems Modeling Language*, 3rd ed.; Morgan Kaufman/The OMG Press: Burlington, MA, USA; Elsevier Science & Technology: Amsterdam, The Netherlands, 2014; ISBN 978-0-12-800202-5.
7. Rabiner, L.R. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proc. IEEE* **1989**, *772*, 257–286. [CrossRef]
8. Hansen, E. Solving POMDPs by searching in policy space. In Proceedings of the Fourteenth International Conference on Uncertainty in Artificial Intelligence (UIA-98), Madison, WI, USA, 24–26 July 1998.

9. Siaterlis, C.; Genge, B. Cyber-Physical Testbeds. *Commun. ACM* **2014**, *57*, 64–73. [CrossRef]
10. Madni, A.M.; Sievers, M. SoS Integration: Key Considerations and Challenges. *Syst. Eng.* **2014**, *17*, 330–347. [CrossRef]
11. Bernijazoo, R.; Hillebrand, M.; Bremer, C.; Kaiser, L.; Dumitrescu, R. Specification Technique for Virtual Testbeds in Space Robotics. *Procedia Manuf.* **2018**, *24*, 271–277. [CrossRef]
12. Denno, P.; Thurman, T.; Mettenburg, J.; Hardy, D. On enabling a model-based systems engineering discipline. *INCOSE Int. Symp.* **2008**, *18*, 827–845. [CrossRef]
13. Russmeier, N.; Lamin, A.; Hann, A. A Generic Testbed for Simulation and Physical Based Testing of Maritime Cyber-Physical System of Systems. *J. Phys. Conf. Ser.* **2019**, *1357*, 012025. [CrossRef]
14. Brinkman, M.; Hahn, A. Physical Testbed for Highly Automated and Autonomous Vessels. In Proceedings of the 16th International Conference on Computer and IT Applications in the Maritime Industries, Cardiff, UK, 15–17 May 2017.
15. Fortier, P.J.; Michel, H.E. Computer Systems Performance Evaluation and Prediction, Chapter 10. In *Hardware Testbeds, Instrumentation, Measurement, Data Extraction and Analysis*; Digital Press; Elsevier Science & Technology: Amsterdam, The Netherlands, 2003; pp. 305–330.
16. Bestavros, A. Towards safe and scalable cyber-physical systems. In Proceedings of the NSF Workshop on CPS, Austin, TX, USA, 16 October 2006.
17. Madni, A.M.; Erwin, D.; Sievers, M. Architecting for Systems Resilience: Challenges, Concepts, Formal Methods, and Illustrative Examples. *MDPI Systems*. to be published in 2021.
18. Kellerman, C. *Cyber-Physical Systems Testbed Design Concepts*; NIST: Gaithersburg, MD, USA, 2016.
19. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms. *arXiv* **2017**, arXiv:1707.06347.
20. Haarnoja, T.; Zhou, A.; Hartikainen, K.; Tucker, G.; Ha, S.; Tan, J.; Kumar, V.; Zhu, H.; Gupta, A.; Abbeel, P. Soft actor-critic algorithms and applications. *arXiv* **2018**, arXiv:1812.05905.
21. Torabi, F.; Warnell, G.; Stone, P. Behavioral cloning from observation. *arXiv* **2018**, arXiv:1805.01954.
22. Ho, J.; Ermon, S. Generative adversarial imitation learning. *arXiv* **2016**, arXiv:1606.03476.
23. Juliani, A.; Berges, V.; Teng, E.; Cohen, A.; Harper, J.; Elion, C.; Goy, C.; Gao, Y.; Henry, H.; Mattar, M.; et al. Unity: A General Platform for Intelligent Agents. *arXiv* **2020**, arXiv:1809.02627. Available online: https://arxiv.org/abs/1809.02627 (accessed on 4 March 2021).
24. Madni, A.M.; Lin, W.; Madni, C.C. IDEON™: An Extensible Ontology for Designing, Integrating, and Managing Collaborative Distributed Enterprises in Systems Engineering. *Syst. Eng.* **2001**, *4*, 35–48. [CrossRef]
25. Madni, A.M. Exploiting Augmented Intelligence in Systems Engineering and Engineered Systems. *Syst. Eng.* **2020**, *23*, 31–36. [CrossRef]
26. Madni, A.M.; Sage, A.; Madni, C.C. Infusion of Cognitive Engineering into Systems Engineering Processes and Practices. In Proceedings of the 2005 IEEE International Conference on Systems, Man, and Cybernetics, Hawaii, HI, USA, 10–12 October 2005.
27. Madni, A.M. HUMANE: A Designer's Assistant for Modeling and Evaluating Function Allocation Options. In Proceedings of the Advanced Manufacturing and Automated Systems Conference, Louisville, KY, USA, 16–18 August 1988; pp. 291–302.
28. Trujillo, A.; Madni, A.M. Exploration of MBSE Methods for Inheritance and Design Reuse in Space Missions. In Proceedings of the 2020 Conference on Systems Engineering Research, Redondo Beach, CA, USA, 19–21 March 2020.
29. Madni, A.M. Integrating Humans with Software and Systems: Technical Challenges and a Research Agenda. *Syst. Eng.* **2010**, *13*, 232–245. [CrossRef]
30. Shahbakhti, M.; Li, J.; Hedrick, J.K. Early model-based verification of automotive control system implementation. In Proceedings of the 2012 American Control Conference (ACC), Montreal, QC, USA, 27–29 June 2012; pp. 3587–3592.