Event-driven Multi-Agent Concurrent and Collaborative Coordination

Pedro Acevedo
Department of Computer Science
University of North Carolina Wilmington
Wilmington, North Carolina, USA
acevedop@uncw.edu

Fengze Zhang
School of Applied and Creative Computing
Purdue University
West Lafayete, Indiana, USA
zhan5455@purdue.edu

Christos Mousas
School of Applied and Creative Computing
Purdue University
West Lafayette, Indiana, USA
cmousas@purdue.edu

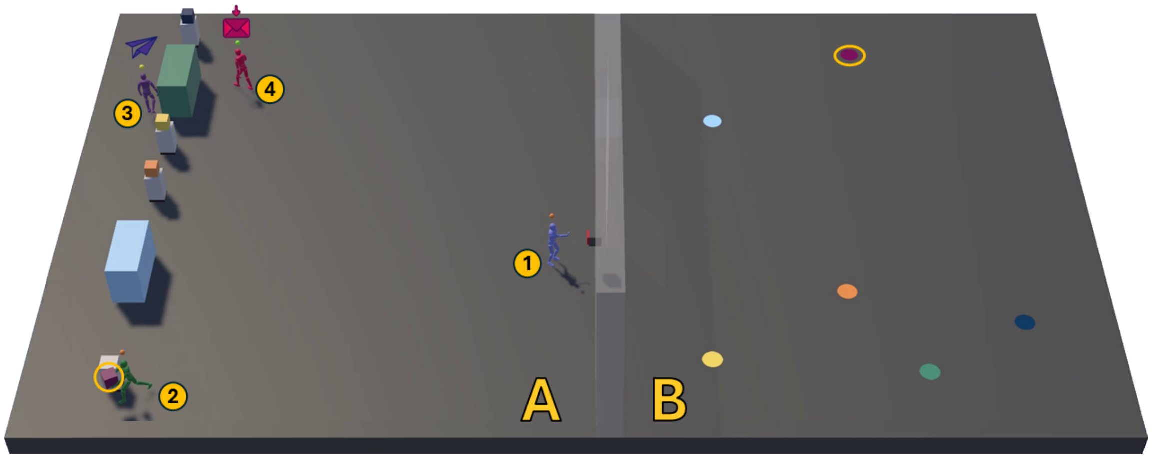


Figure 1: Event-driven behavior tree (EDBT) agents interact within the simulation to complete a moving task. (1) Agent 1 is pressing a button to open the door separating Room A and Room B. (2) Agent 2 has grabbed a box that needs to be placed at its final destination in Room B. (3) Agent 3 has sent a request for assistance to move a large object. (4) Agent 4 has received the message and is preparing to help Agent 3 complete the task.

Abstract

Multi-agent systems (MAS) model the interaction of multiple autonomous agents within a shared environment to achieve a common goal. These agents operate independently, considering the output of other agents to make decisions that contribute to the simulation's objectives. MAS can create dynamic and nondeterministic simulations where agents must adapt and collaborate to optimize task completion. Coordination in MAS remains an open challenge, requiring effective strategies that account for agent behaviors, environment interactions, and overall objectives. Building on prior work in behavior tree (BT) coordination (Coord-EDBT) and the smart objects paradigm, we present a simulation approach that enables agents to use a messaging protocol for sending and receiving requests, facilitating collaborative task execution. We validate our system through a collaborative moving task in which multiple agents transport objects from Room A to Room B. We described our implementation, including behavior trees, environment settings, and object interactions, and illustrated its capabilities through multiple examples. An initial evaluation revealed that the request-based communication mechanism enhanced agent collaboration, supporting more effective coordination in dynamic simulation environments. Our system and demos can be found at [URL is omitted due to double-blind review process].

CCS Concepts

• **Computing methodologies** → **Multi-agent systems**; **Animation**.

Keywords

Multi-agent Systems, Behavior Tree, Coordination, Simulation, Autonomous Agents

ACM Reference Format:
Pedro Acevedo, Fengze Zhang, and Christos Mousas. 2025. Event-driven Multi-Agent Concurrent and Collaborative Coordination. In *The 20th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry (VRCAI '25), December 13–14, 2025, Macau, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3779232.3779273

1 Introduction

Multi-agent systems (MAS) have been a topic of research interest due to their potential for modeling distributed autonomous entities [Amirkhani and Barshooi 2022; Chen et al. 2023; Gronauer and Diepold 2022; Viqueira and Cousins 2019]. A MAS is a system of multiple distributed agents that make decisions autonomously based on a set of predefined actions and interact within a shared, partially controlled environment [Weiss 1999]. These agents aim to achieve specific goals using intelligent and reactive behaviors. When agents operate within dynamic and nondeterministic simulations, coordination and collaboration become essential [Shoulson et al. 2013]. Applications of MAS span various domains, including non-player character (NPC) behavior in games [Agis et al. 2020], autonomous driving [Shalev-Shwartz et al. 2016], social roles [Zhang et al. 2019], and robotics [Aloor et al. 2024]. Exploring how to coordinate MAS highlights its relevance for future applications, integrating these agents. Depending on the system's objectives, various techniques can be used to model agent behavior. These include decision-making algorithms, reinforcement learning, and behavior trees (BTs) [Iovino et al. 2022]. Equally important is the role of the environment, which imposes constraints such as physics, motion, and object interactions [Albrecht et al. 2024]. Among these techniques, BTs provide a structured approach for defining agent behavior, serving as a *"control block that guarantees the execution of a specific chain of actions given a specific set of perceptions"* [Pereira and Engel 2015]. Initially developed for the video game industry as an alternative to finite state machines (FSMs), BTs have since been applied in robotics, control theory, and general agent modeling. Unlike learning-based approaches, BTs ensure reliability, as agents execute behaviors exactly as designed by experts. Additionally, BTs facilitate the construction and maintenance of complex models while preserving readability.

In MAS, behavior modeling techniques face additional challenges due to agents sharing an environment and resources [Dorri et al. 2018]. The need for coordination arises because agents must consider the actions of other agents when making decisions, thereby increasing complexity and potentially causing conflicts. Without coordination mechanisms, MAS can become unmanageable. Several approaches have been developed to address this issue [Albrecht et al. 2024; Lowe et al. 2017; Shang et al. 2023; Wang et al. 2016]. Specifically, utilizing BTs, Agis et al. [Agis et al. 2020] developed an extension called Coord-EBT. Their method included a messaging protocol by integrating new types of nodes for coordination. The nodes enabled agent communication through a request protocol. Building on this work in coordination [Agis et al. 2020] and smart objects paradigm [Abaci et al. 2005], we designed a simulation using BT-driven agents that collaborate on a common task. We incorporated smart objects with properties such as concurrency, goal constraints, preconditions, and state changes to enrich the simulation environment. We validate our framework using a collaborative moving task, where multiple agents must transport objects from room A to room B (see Figure 1). Some objects require multiple agents to move (e.g., a large object), while others depend on specific preconditions (e.g., opening a door between rooms). All agents shared the same event-driven behavior tree (EDBT) to operate within the simulation. Overall, our contributions are the following:

- we designed a multi-agent simulation integrating an existing BT extension and smart objects for dynamic coordination environments;
- we introduced smart objects with properties such as concurrency and preconditions to enhance complex simulation scenarios; and
- we evaluate our system through multiple example scenarios, demonstrating how agents coordinate to complete a moving task.

We organized our paper as follows. In Section 2, we discuss work related to our project. In Section 3, we present the background of our EDBT, Coord-EBT, and smart objects paradigm. In Section 4, we detail the methodology for integrating Coord-EBT with smart objects to handle multi-agent scenarios that demand collaboration and agent-object interactions. In Section 5, we present various simulated moving task scenarios to empirically validate the proposed MAS. In Section 6, we present the evaluation of the proposed approach. Last, in Section 7, we conclude and discuss potential future directions.

2 Related Works

2.1 BT-driven Agents

BTs allow agents to handle complex interactions while maintaining scalability and reusability. Beyond gaming, BTs have been extended for diverse use cases [Iovino et al. 2022]. Liu et al. [Liu et al. 2023] used BT-driven AI agents to annotate game-level collaboration degrees for synthesizing a virtual reality experience. Zhang et al. [Zhang et al. 2021] employed BTs to simulate agents for workspace and workplan optimization, including use cases such as a fast-food kitchen and a supermarket. Regarding the extension of BTs, Neupane and Goodrich [Neupane and Goodrich 2019] combine the BT with a grammatical evolution approach to model the behavior of different swarms of agents, thereby replicating animal behavior. Johansson and Dell'Acqua [Johansson and Dell'Acqua 2012] extended BT to reactively adapt actions based on emotions, considering factors such as time-discounting, risk perception, and planning. In robotics, Colledanchise et al. [Colledanchise et al. 2019] developed a planning approach to automatically create and update a BT, enabling the control of a robot in a dynamic environment. BT research has focused on creating extensions to enable various possible agent behaviors, such as coordination with multiple agents [Agis et al. 2020], parallel actions [Colledanchise and Natale 2018], and learning methods [Colledanchise et al. 2014; Sprague and Ogren 2022]. In this work, we adopted an extended version of BTs to model agent behavior and defined a specific action set that enables collaborative interactions among agents to achieve a shared goal.

2.2 Multi-Agent Systems

MAS can be applied to multiple real-world scenarios, considering how multiple agents/actors interact with each other to accomplish the same goal. Researchers have focused on performing and modeling the complex task of agent coordination in different fields. For

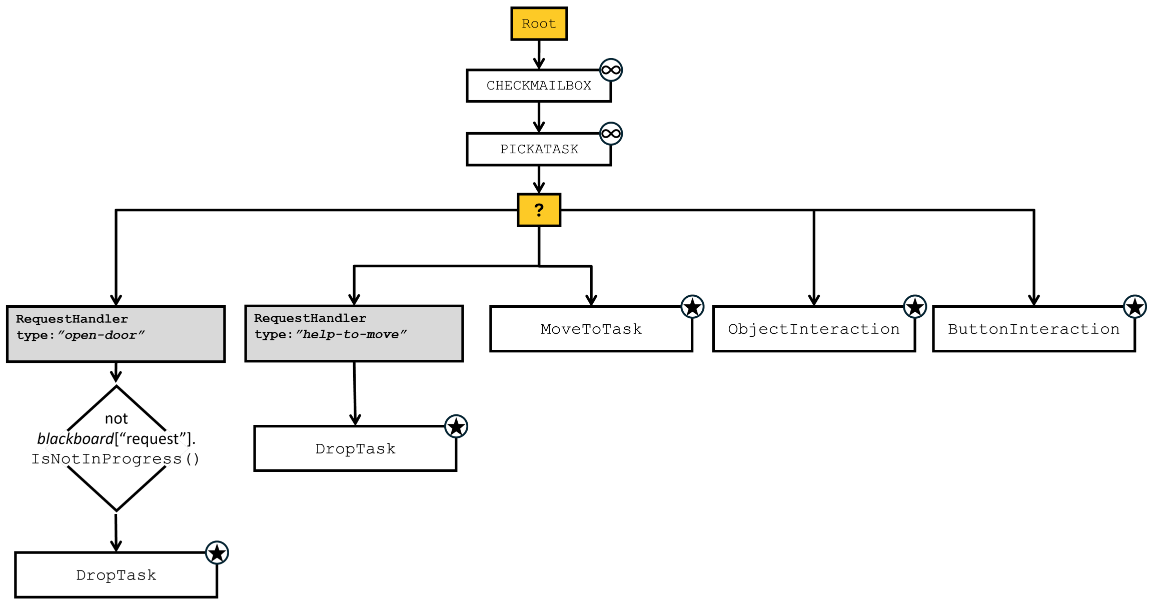

Root
CHECKMAILBOX
PICKATASK
?
RequestHandler type:"open-door"
RequestHandler type:"help-to-move"
MoveToTask
ObjectInteraction
ButtonInteraction
not blackboard["request"]. IsNotInProgress()
DropTask
DropTask

Figure 2: The agent's EDBT. The nodes are represented as rectangles and symbols at the top-right corner, indicating the type of node. Service nodes are represented by an infinity (∞) symbol, selection nodes with question marks, and leaf action nodes by a star (⋆). The gray background rectangles represent a blackboard observer decorator (BOD) node proposed by Agis et al. [Agis et al. 2020].

instance, Shoulson et al. [Shoulson et al. 2013] designed an event-centric planning framework based on BTs for directing interactive narratives in complex 3D environments populated by virtual humans. Their approach demonstrated agent coordination in a prison break scenario, where agents assumed roles such as robbers and police. Agis et al. [Agis et al. 2020] proposed Coord-EBT, a method for agent coordination based on BTs. They included a messaging protocol that allows agents to send requests to complete multiple collaborative tasks in a simulation environment. Their implementation demonstrated how agents could collaborate as firefighters to extinguish fires. Shang et al. [Shang et al. 2023] proposed a reinforcement learning approach for agent collaboration tasks using a constraint-based method. Their approach divides the policy into phases, each treated as a constraint to guide the task objective. They validate their method using a tray-balancing task, where two agents must first coordinate to keep the tray stable (Phase 1) and then work together to place a moving target at its final destination (Phase 2). Liu et al. [Liu et al. 2019] investigated the emergence of cooperative behaviors in multi-agent competitive environments using a population-based reinforcement learning approach. They implemented a two-versus-two soccer simulation, where agents compete in groups to score goals. The agents were controlled using physics-based motions, including forward and backward movement, rotational torque, and jumping forces. Their methods demonstrated how agents develop coordinated strategies in response to competition, highlighting the potential for emergent collaboration in reinforcement learning settings. In this work, we adopt the BT approach, specifically leveraging the Coord-EBT framework. We define multiple collaborative tasks in which agents must communicate to achieve their goals, thereby facilitating coordination in dynamic simulation environments.

3 Background

3.1 Event-driven Behavior Tree

An EDBT is a directed tree that encapsulates agents' actions sorted on sequence and conditions. EDBT extends traditional BT by incorporating event-based mechanisms, allowing nodes to be triggered by external or internal events instead of being executed in a fixed order [Champandard and Dunstan 2019]. The tree's root represents the starting point of executions; those executions or signals are denominated `Ticks`, which are called on a specific frequency. These `Ticks` are propagated on the tree, enabling the execution of the actions in each of the children's nodes, starting from the most left-side node. A node will be only executed when receiving `Ticks`. This child node will return a `Running` state to the parent, wherever the actions are being executed, a `Success` state once the goal is achieved, or a `Failure` state otherwise. A single node or leaf node is referred to as a primitive behavior. Composed behaviors utilize a combination of primitive behaviors and other composed behaviors, thereby defining a behavior hierarchy. Additionally, EDBT has a blackboard. A blackboard is a data structure composed of a dictionary, which is a collection of (`key`, `value`) pairs. Those blackboards are private; no values or key information are shared between trees.

All nodes, excluding the root, can be categorized as:

- **Action:** These nodes execute commands to the agent when receiving `Ticks`, such as grabbing an object. Once the action is completed, it returns `Success`; otherwise, it returns

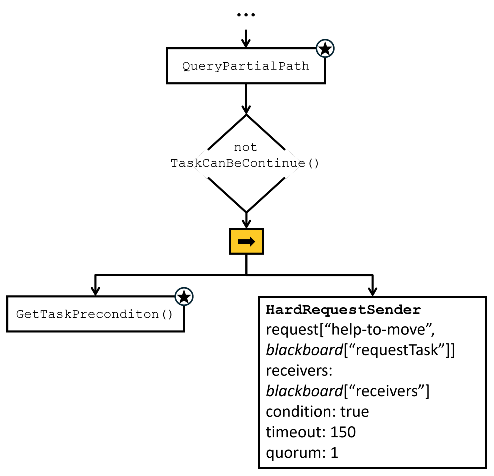


Figure 3: Checking for a complete path when moving a box to the next room. The agent will send HRS if it is not possible.

Failure. When the action is ongoing, it returns Running. Shaded boxes represent actions (see Figure 2, DropTask node).

- **Composite:** These nodes propagate the Ticks directly to their children in a specific order. The node decides which child will be activated and the return state. These nodes can have multiple child nodes. The composite is represented as a square symbol or rectangle with text. For Composite nodes, there are common types such as:
  - **Sequences:** These nodes are used when actions need to be executed in order, one after the other, and their execution depends on the completion of the previous one. The node routes the Ticks to the left child and returns Running or Failure based on the node's state. The node will return to Success if and only if all children return Success. The child nodes will be executed in order; once a node returns Success, the subsequent node will receive the Ticks; otherwise, in the Failure state, the subsequent nodes will not be executed. The symbol of the Sequence node is a box labeled with "$-$ >" (see Figure 3).
  - **Fallback:** These nodes are used when multiple actions aim at the same goal, and their Success is independent of each other. The node will route the Ticks to the left-side child and return Success or Running based on the current child. A Fallback node returns Failure if and only if all children fail. Once a child node returns Running or Success, the Ticks will not be routed to other children. The symbol of the Fallback node is a box labeled with "?" (see Figure 2).
  - **Parallel:** These nodes are used when all children's nodes must be executed simultaneously. Considering M as nodes and N as a children count and $M < N$, if M children return Success, then the Parallel node will return Success too. If more than $N - M$ nodes return Failure, then this node fails. Otherwise, the node will return a Running status. The symbol of the Parallel node is a box labeled with "=>."
- **Decorator:** These nodes have a single composite or action node as a child. Some examples include conditional nodes, which check whether specific constraints and situations are met to propagate Ticks to their children. They return Success and Failure when the conditions are true or false, respectively. The conditions are represented by a diamond shape (see Figure 2). The observer decorator is used to react to events and abort nodes that are in the Running status. Additionally, the blackboard observer decorator (BOD), which checks a blackboard key and has an abortion rule, is set to state Running based on whether the condition of the blackboard key is met. Still, if the abortion rule happens, the node execution can be stopped.
- **Service:** These nodes change the behavior of their composite node by adding a method and altering the activation frequency. Whenever a service node receives a Tick, it sends the Tick to its child and repeatedly calls the method at the specified frequency as long as at least one of the composite's descendants returns a Running status. Then, once its child returns a Success/Failure status, the service node returns the result to its parent. Since the method called by a service node is executed concurrently, nodes in the Running status are not interrupted. Services have a single child and are represented graphically by the infinity (∞) symbol (see Figure 2, PickATask node).

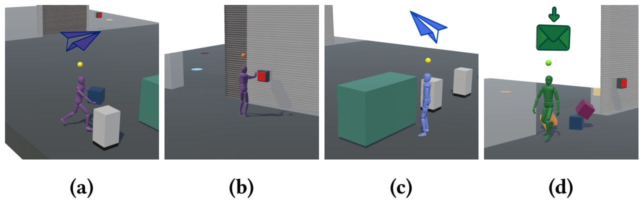


Figure 4: Agents in a scenario that demands coordination. (a) An agent has an object in its hand but is *waiting* (represented by a yellow sphere) for another agent to open the door. (b) A second agent gets a request and presses a button to open the door. (c) An agent needs to move an object that is too large for a single agent to handle, and a second agent has dropped its current task and is ready to assist the other agent in moving the large object.

3.2 Coord-EBT

Coord-EBT is an EDBT extension incorporating a messaging protocol for multi-agent coordination. Agis et al. [Agis et al. 2020] introduced the Coordination nodes, which manage message exchanges between agents. Within this protocol, a sender agent (s) can send a message to another agent (r) to execute a task or a specific sub-tree within its BT. The message is a 4-tuple:

$$msg = (s,\ req,\ c,\ t)\,, \tag{1}$$

where s is the sender of the message, req denotes the specific request, c denotes a condition that needs to be met to be able to execute req, and t denotes a time limit after which the message is discarded if not acted upon. A request is a pair:

$$req = [type, \ parameters], \tag{2}$$

where *type* specifies the branch that will be executed by r and *parameters* as a tuple of inputs needed for the action (e.g., $req =$ [“$open - door$, ”(button position)]). The messages are stored in the receiver Mailbox, which is a priority queue for any message sent between agents. Each agent has a service node (CheckMailbox) that continuously checks for new messages in their Mailbox and processes them upon arrival.

In response to requests, there is the Request Handler (RH), which encapsulates request types or task branches that should be executed when a message is accepted, as callbacks. RH nodes consist of:

- a blackboard key that tracks whether a request of a specific type has been received;
- a condition defining when the request should be processed; and
- an aborting rule that terminates the request if necessary.

There are two types of request nodes:

- **Soft Request Sender (SRS):** The sender continues executing its BT after sending a message; and
- **Hard Request Sender (HRS):** Executing both the sender's and receiver's subtrees requires a quorum q (a minimum number of confirmed receivers) before proceeding.

Unlike traditional BT approaches that rely on a shared blackboard for coordination, Coord-EBT enables reactive, message-driven coordination, ensuring dynamic agent interactions without persistent shared memory dependencies.

3.3 Smart Object Paradigm

Smart objects are instances of objects in the environment that store information, which can be shared with agents upon direct interaction. This enables agents to dynamically adapt their actions based on the object's properties and constraints, effectively allowing them to “learn” how to handle interactions with encountered objects [Abaci et al. 2005]. Beyond their geometric attributes for visualization, smart objects encapsulate semantic information that aids animation and interaction. These objects store data as a set of attributes, conveying details such as key interaction points (e.g., where and how a virtual character should position its hands to grasp the object), predefined animation sequences (e.g., a door opening), and non-geometric properties (e.g., weight or material type). This semantic information enables virtual characters to perform context-aware interactions, such as grasping, moving, or operating objects (e.g., machines or elevators). Additionally, smart objects facilitate action planning by incorporating *preconditions*—must be met before an action can be executed—and *effects*, which define the changes to the scene state after the action is performed.

4 Methodology

In this section, we present the methodology for integrating Coord-EBT with smart objects to handle multi-agent scenarios that demand collaboration and agent-object interactions. In this process, we define the different subtree structures and object components to create a simulation. For our implementation, we focused on a simulation in which agents must coordinate on a moving task, as illustrated in Figure 4, involving the placement of smart objects from one room to another, encompassing preconditions, concurrency, and coordination.

4.1 Environment Specification

We simulated a moving task scenario involving n agents, each with the same role and EDBT. The scene comprises two rooms (i.e., A and B) and a door connecting them. In this environment, the agents operate autonomously, aiming to complete all tasks in the task pool (see Section 4.2). All agents share the same set of capabilities, including grabbing objects, moving to a destination, checking their Mailbox, sending requests, and pressing buttons. Additionally, agents are aware of each other's presence in the scene for navigation purposes. They dynamically recalculate their paths whenever they detect other agents or obstacles in their way. The environment is simulated inside the physics simulator and NavMesh components of the Unity game engine.

4.2 Task Pool

The task pool is a structure that compiles all task-related information within the environment. A task is defined as an action that agents must perform on a smart object, such as delivery or pressing a button. Each task can exist in one of three states: *queue*, in *progress*, or *done*. A task can also be classified as non-interruptible, meaning that the task cannot be dropped. The task pool is shared among agents, as it represents the current state of the simulation. Additionally, it establishes dependencies between tasks through preconditions, where a specific task x can only begin (or is in *progress*) once another task y has been completed (i.e., *done*). The pool filters available tasks, ensuring that agents select only those still in the *queue* state. The simulation concludes when all tasks have been marked as *done*.

4.3 Smart Objects

These objects store semantic and animation-related information that agents can access upon interaction. For example, they include the object state, which can be *initial*, *agent*, referring to when an agent is grabbing the object, *goal*, once the agent reaches the final destination, and *shared*, which is when multiple agents hold the object. They also store data such as the final destination, grabbing points for animation, type, and objective.

In this scenario, we defined two main types of smart objects: *box* and *button*. A *button* serves the primary function of activating a sliding door, which opens after a specific delay. The button must remain pressed until the door is fully open. Once opened, the door remains open for the rest of the simulation.

Box is the central component of the simulation. Each box has a designated destination in Room B and stores information about its final position, animation control, and box type. There are three box types: *regular*, *concurrent*, and *large*. Specifically:

- a *regular box* can be carried by a single agent and transported to its final position;
- a *concurrent box* can be moved alongside other boxes, allowing an agent to perform multiple tasks simultaneously when handling concurrent or regular boxes. Additionally, these

boxes have extra capacity, enabling another box to be placed on top of them; and

- a *large box* requires coordination between two or more agents to be transported to its destination.

This classification of different cases requires more agent-object interactions, considering the object inherits actions to complete the task.

4.4 Task Interaction

In our implementation, all agents perform the same EDBT (see Figure 2). These entities are configured as 3D virtual characters that follow commands based on the task pool. An agent operates in one of four states: *available*, *occupied*, *follow*, or *waiting*. Specifically:

- an agent is *available* when it has no assigned task and it is represented with green color;
- an agent becomes *occupied* once it starts a task that is still in progress, and is represented with orange color;
- an agent enters a *follow* state once an interaction between a large object happens, and two or more agents move the object, which is represented in blue; and
- an agent enters a *waiting* state when its assigned task has a precondition to be completed before it begins, and when an agent is expecting to get a response to a sent request. This state is represented by the yellow color.

Furthermore, an agent possesses a property called "power," which is related to the number of objects an agent can move. In the case of a small object, a total power of one is required, but for large boxes, which require a power of two or higher, a single agent may not be sufficient. Agents, in an *available* state, repeatedly check incoming messages through the service `CheckMailbox` and continuously seek tasks using the `PickATask` service node, which queries for tasks that are in the queue. In addition, agents have the respective request handler for the coordination task *"open-door"* and *"help-to-move"* (see Section 4.6). Once an agent selects a task, it proceeds to complete it. The task's status is then updated to *progress* and reflected in the shared task pool, which all agents can access as part of the environment. The agent then executes **Behavior 1** related to the node `MoveToTask`.

Behavior 1 (Move to task)

(1) The agent selects a task x and sets it up as in progress.
(2) The agent retrieves the position of x, which corresponds to a smart object obj.
(3) The agent determines an appropriate interaction point in the environment. This point is provided by obj, which maintains multiple interaction points and assigns one to each requesting agent.
(4) Using a navigation mechanism and obstacle avoidance, the agent moves to the assigned interaction point.
(5) Once the agent reaches the interaction point, it begins executing the task x by interacting with obj.

`ObjectInteraction` and `ButtonInteraction` are other action nodes describing specific behaviors related to smart object manipulation (see Sections 4.4 and 4.5, respectively). Among other behaviors, an agent can drop a task. If it needs to respond to a request or event, it must drop its current task (not between concurrent tasks) before performing another action. An agent follows **Behavior 2** related to the node `DropTask`.

Behavior 2 (Drop a task)

(1) The agent is executing the task x but must abandon it to perform another task.
(2) The agent returns the task x to the *queue* state.
(3) The agent detaches or releases all associated smart objects obj and pending objects.
(4) All smart objects obj, including the ones from the pending list, remain in their current position.
(5) The agent changes its state to *available*.
(6) The agent resumes EDBT execution.

4.5 Object Interaction

The agent interacts with a smart object as part of a task. This process begins once the agent reaches the designated interaction point and initiates the task (see **Behavior 3**). A task will be set up as in *progress* as soon as it is assigned and the agents start to move to the interaction point. This behavior applies to all object types, with specific interactions for boxes, including handling large objects and concurrent tasks (see Figure 5).

Behavior 3 (Smart object interaction)

(1) The agent reaches a smart object obj.
(2) The agent queries obj for interaction details, including contact points (e.g., where the agent should place its hands) and object type.
(3) Based on obj's type, the agent performs the corresponding animation.
(4) The agent continues executing its EDBT, following the task's requirements.

When interacting with a box, the agent first validates whether the box needs to be moved and whether it is *available* to do so. The agent's actions depend on the box type. For a *regular box*, the agent performs a grab action, establishes the interaction, and checks for a path to the next room. For a large box (see Figure 6), the agent verifies whether it has sufficient power to move it. If not, an **HRS** is triggered to request assistance. Furthermore, the agent validates if the box can be moved by checking for a path to the next room `QueryPathPartial` (see Figure 3). If the path is partial or not complete, the agent will send a **HRS** for help.

For a *concurrent box* (see Figure 10), the agent checks whether concurrent tasks are *available* in the task pool. These tasks involve objects with the concurrent property enabled, allowing agents to reactively grab multiple objects at once. For example, an agent already holding a box and in *waiting* state, may pick up a second one, enabling simultaneous task completion.

4.6 Button Interaction

The agent-button interaction is mapped on the `ButtonInteraction` composite node (see Figure 8). The agent will query the current distance between its position and the button position to perform

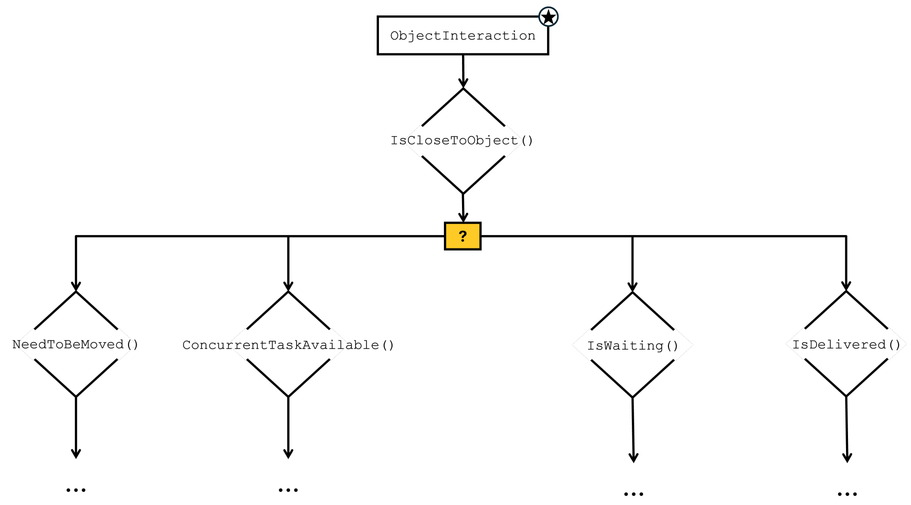


Figure 5: Box interaction hierarchy, considering scenarios such as `NeedToBeMoved`, which determines if a box is available for interaction; `ConcurrentTaskAvailable`, which checks whether another task involves an object marked as concurrent; `IsWaiting`, which validates whether the agent can proceed with the task while in a waiting state; and `IsDelivered`, which marks the task as complete once the agent reaches the final position of the box.

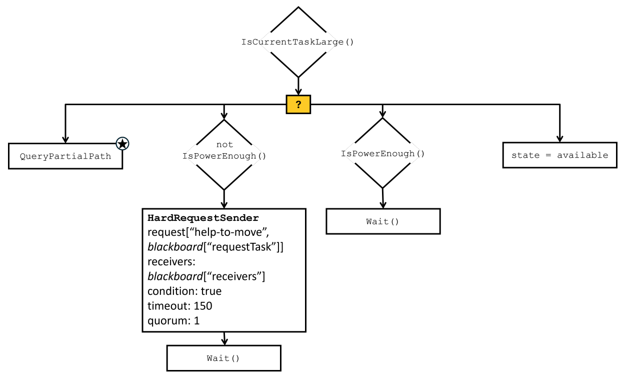


Figure 6: The task is with a large object. The agent will send an HRS.

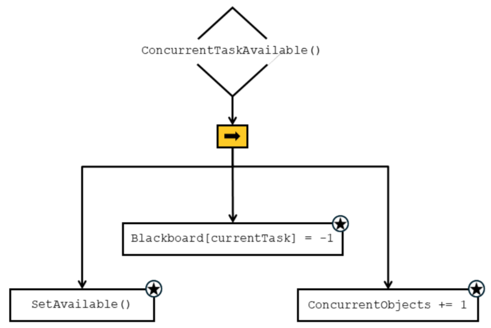


Figure 7: The concurrency behavior.

Behavior 2. Once the agent presses the button, the sliding door will start to open, and the agent remains in place, holding the button until the door is fully open. Once the door is open, the task is set to *done*.

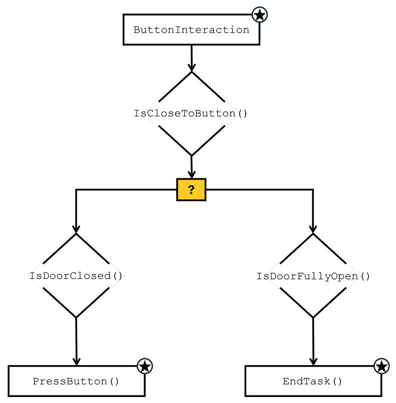
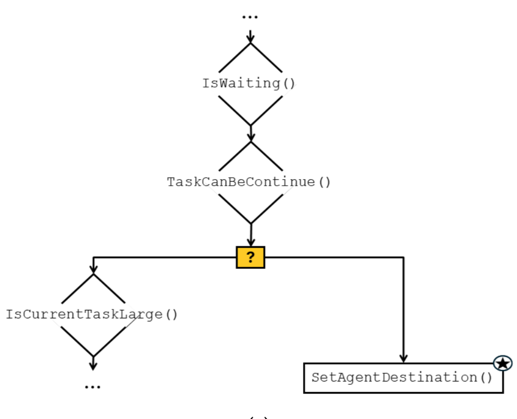


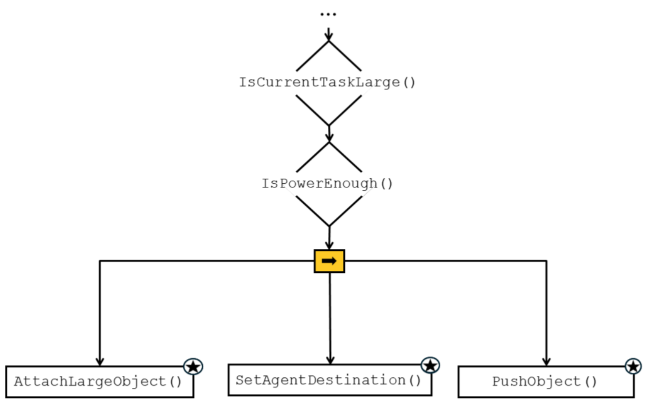
Figure 8: Button interaction behavior hierarchy.

4.7 Coordination

In the defined MAS, there are opportunities for agents to coordinate to complete a task. In this case, agents will use the **HRS** mechanism because it forces agents to wait for prerequisites. For this mechanism, as described by Agis et al. [Agis et al. 2020], agents should perform specific behaviors that leverage this communication interaction (see Appendix **??**). Initially, agents can get requests and execute the behavior associated with the type. Then, an **HRS** is executed after both parties (according to the defined quorum) confirm their agreement (see Figure 9). In this implementation, an agent can send a request r of two types: *"open-door"* and *"help-to-move."*



(a)



(b)

Figure 9: Agents' reactive actions: (a) once an agent is in a *waiting* state and it is checking for the precondition of the current task to be completed before proceeding, and (b) checking whether the task with a large box can be completed.

4.7.1 Opening Door. In this coordination scenario, the agent s picked a task that required moving a box to the next room. This task has an assigned precondition stating that the task of opening the door should be done first before this task can be completed (see Figure 9a). Then, **Behavior 4** is executed.

Behavior 4 (Open door coordination)

(1) Agent s executes **Behavior Handling Hard Request** with a *req* of type *"open-door,"* $q = 1$, and the precondition of not being at the door already.
(2) Agent r, who is focusing on another task and meeting the criteria, sends the confirmation to s:
  (a) The agent r executes **Behavior 2** and will perform the behavior associated with the request type *"open-door"* request.
  (b) The agent r proceeds to execute **Behavior 3** with the button smart object (see Section 4.5).
  (c) Meanwhile, since the agent r is taking the message from s, s enters a *waiting* state and holds the box.
  (d) During each simulation tick, the agent s continuously checks for an *available* path to the box's final destination.
  (e) Once a valid path is detected, the Agent s transitions from a *waiting* state to an *occupied* state and begins moving toward the destination.
  (f) Agent s reaches the final destination, and the task is set up as *done.*
  (g) Agent s change its state to *available.*
(3) None of the agents met the criteria, or the quorum is never met before t elapses:
  (a) Agent s performs **Behavior 4** after a timeout to resend the request.

4.7.2 Large Object. In this coordination scenario, the agent s picked a task that required moving a large box to the next room. Since the box is too large for a single agent to carry (due to insufficient power), the task requires coordination between two or more agents (see Figure 9b). The sequence of events for completing this coordination process is shown on **Behavior 5**.

Behavior 5 (Large object coordination)

(1) Agent s executes **Behavior Handling Hard Request** with a *req* of type *"help-to-move,"* $q = 1$, and the precondition of not being at the door already.
(2) Agent r, who is focusing on another task and meeting the criteria, sends the confirmation to s:
  (a) Agent r executes **Behavior 2** and will perform the **RH** associated with the request type *"help-to-move."*
  (b) Agent r executes **Behavior 3** with the large box smart object (see Section 4.4).
  (c) Meanwhile, since Agent r has accepted the request, Agent s put itself in a *waiting* state.
  (d) At the large object, Agents s and r position themselves on the respective interaction points, adjusting their animations to grasp the object correctly.
  (e) Agent r and s start pushing the large object to the final destination, following a similar path.
  (f) Agents s and r reach the final destination, and the task is set up as *done.*
(3) None of the agents met the criteria, or the quorum is never met before t elapses:
  (a) Agent s performs **Behavior 5** after a timeout to send out back the request.

5 Application Example

In this section, we present various simulated moving task scenarios to empirically validate the proposed MAS.

5.1 Example 1: Open the Door

In this scenario, there are two agents, $A1$ and $A2$, and three tasks: moving two regular boxes ($T1$ and $T2$) and pressing a button to open a door between rooms ($T3$). Agents $A1$ and $A2$ each pick a task related to boxes $T1$ and $T2$, respectively. Once the simulation begins, agents are activated one after another, with the first agent reaching its assigned box first. Upon interacting with $T1$, the agent $A1$ checks for any preconditions and determines that the door must be open before completing the task. Since $T3$ must be completed first, agent $A1$ activates **Behavior 4** to request assistance moving the box. Meanwhile, agent $A2$ reaches its assigned box, but before it can send a request, its `Mailbox` receives a request from agent $A1$. Since agent $A2$ meets the criteria to perform $T3$, drops $T2$, and proceeds to complete the task by pressing the button to open the door. At this stage, the agent $A1$ waits with the box in hand while agent $A2$ stands next to the door, pressing the button as the sliding door opens. Once the door is open enough for the agent $A1$ to proceed, it moves to the final destination to complete the $T1$ task. Once the door is fully open, agent $A2$ becomes *available* and picks up task $T2$, which is the only remaining task in the queue. Agent $A2$ then executes the `ObjectInteraction` node to complete $T2$, and the simulation concludes.

5.2 Example 2: Large Object Interaction

In this scenario, there are two agents, $A1$ and $A2$, and two tasks: pressing a button to open a door ($T1$) and moving a large box ($T2$). Each agent selects a task from the task pool. Agent $A2$ is assigned $T1$ and moves to the button to open the door, while agent $A1$, assigned $T2$, heads toward the large box. Upon reaching the box, agent $A1$ evaluates whether it has sufficient strength to move it. Since agent $A1$ has a *power* of one, but the box requires a total of two, it determines that assistance is needed and initiates **Behavior 5** to request help. Meanwhile, the agent $A2$, positioned at the button, begins pressing it to open the sliding door. Because the door requires the agent to remain in place while it opens, $A2$ cannot move.

At this moment, agent $A2$ receives a *"help-to-move"* request in its `Mailbox`. However, since $T1$ is classified as a non-interruptible task, agent $A2$ continues performing its assigned action without responding to the request. To ensure the request is received, agent $A1$ will resend the request until a response is successful. Once $A2$ completes $T1$, it processes the pending request and moves to assist $A1$, enabling the completion of $T2$.

5.3 Example 3: Coordination Scenario

In this scenario (see Figure 1), there are four agents (i.e., $A1$, $A2$, $A3$, and $A4$) and seven tasks: pressing a button to open a door ($T1$), moving a regular box ($T2$), moving concurrent boxes ($T3$, $T4$, and $T5$), and moving large boxes ($T6$ and $T7$). As tasks are assigned randomly, multiple outcomes are possible. In the following paragraphs, we describe a scenario that illustrates how the EDBT framework operates.

In this scenario, the initial task distribution is as follows: $A1$ is assigned $T1$ (opening the door), $A2$ is assigned $T2$ (moving a regular box), $A3$ takes $T3$ (a concurrent box), and $A4$ handles $T5$ (another concurrent box). While $A1$ is *occupied* with opening the door, the other agents proceed to interact with their respective objects. Since $T3$ and $T5$ are concurrent tasks, $A3$ has the opportunity to pick up a second concurrent box, allowing it to carry two boxes simultaneously.

Once the door fully opens, $A1$ is free to select a new task and picks $T6$, a large object. Meanwhile, the other agents proceed to the next room to place their boxes. When $A1$ reaches the large object, it executes **Behavior 5** and sends a help request. This request is accepted by $A3$, which is still holding two concurrent boxes. In response, $A3$ executes **Behavior 2**, dropping both boxes on the floor before assisting with $T6$. At this point, $A4$, originally assigned $T5$, returns to the room to pick another *available* concurrent task ($T3$) since $A3$ just dropped two boxes. As a result, $A4$ now carries two boxes. Meanwhile, $A2$, having completed $T2$, selects a new task, $T4$, and picks up one of the boxes left by $A3$.

With $A1$ and $A3$ now collaborating to push the large object to its destination, $A2$ and $A4$ complete their respective tasks. Once finished, the final task, $T7$, is assigned to a single *available* agent, who moves toward the smart object in the other room. Since no other tasks remain, the rest of the agents remain idle at their final destinations, *waiting* for further assignments. At this moment, a *"help-to-move"* request is received by $A1$, which moves into position to execute **Behavior 5** once again. With $A1$ and $A2$ working together, the large object is moved, marking $T7$ as completed. This concludes the simulation.

6 Evaluation

We evaluated the effectiveness of the proposed approach by comparing two methods: (1) a baseline method without Coord-EBT capabilities and (2) the current method incorporating the messaging mechanism. It is important to note that the baseline still included the tasks poll and the same smart objects; however, in this version, agents would wait for a fixed timeout for the task's precondition to be met. We set a 10-second timeout for the *waiting* state to allow enough time for another agent to select and complete the prerequisite task, as we noticed during our development and testing process that less than 10 seconds would result in frequent premature timeouts, causing agents to abandon their current tasks unnecessarily and degrading overall coordination efficiency. If not satisfied within that time, the agent would drop the task and select a new one. We defined a scenario consisting of 11 tasks, including door opening, two large objects, four concurrent boxes, and four regular boxes, and varied the number of agents (i.e., 2, 4, 6, 8, and 10). Each configuration was run 10 times. We used the average task completion time as the evaluation metric. A simulation run was considered complete once all tasks were marked as *done*.

As shown in Figure 10, results indicate a consistent decrease in completion time as the number of agents increased, which was expected due to the broader task coverage. Quantitatively, the baseline method showed higher means and larger variability across all agent counts (e.g., 184.14 ± 58.91 s with 2 agents, decreasing to 53.52 ± 16.38 s with 10 agents), whereas the messaging-based approach achieved substantially faster and more stable performance (e.g., 108.95 ± 10.22 s with 2 agents and 34.58 ± 4.10 s with 10 agents).

Across all conditions, our framework outperformed the baseline and exhibited markedly lower standard deviations, demonstrating more consistent and reliable coordination. These results show that integrating Coord-EDBT significantly enhances multi-agent collaboration and overall task completion efficiency.

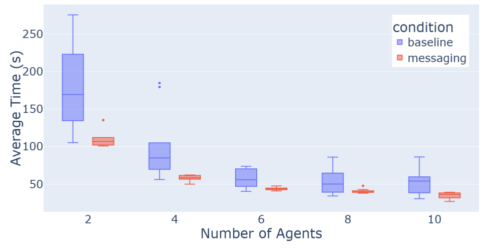


Figure 10: The evaluation results, showing the completion time between the baseline and the messaging conditions.

7 Conclusion and Future Work

In this paper, we describe the implementation of autonomous agent behavior using Coord-EDBT and smart objects in a MAS simulation scenario. We presented our system by detailing different EDBT instances and illustrating how simulation elements interact. To validate our approach, we implemented a collaborative moving task where multiple agents transport objects from Room A to Room B. Various examples demonstrated how collaboration is facilitated through the message/request mechanism, including tasks such as opening doors and moving large objects. Additional scenarios explored concurrency in box handling, varying agent numbers, and multi-interface interactions, highlighting how agents rely on each other to complete tasks. An evaluation showed that our approach outperformed a baseline MAS in most cases, demonstrating the effectiveness of the communication mechanism. While BTs provide structured agent coordination, they also have limitations, particularly in their reliance on expert design and the inherent predictability of agent behavior. However, in this context, we found them suitable as they allow controlled agent collaboration through predefined coordination mechanisms.

For future work, we plan to extend our system by incorporating a failure callback mechanism for handling unsuccessful requests. In the current implementation, when a request fails, the agent attempts to resend it after the message timeout has elapsed. However, instead of continuously retrying, an agent should be able to recognize the failure and select an alternative task that can be completed within the scenario, preventing it from getting stuck. Additionally, we aim to compare our framework with other collaboration mechanisms, such as planners for scene synthesis, and to enhance the simulation environment by incorporating more dynamic elements, including utility-based decision-making, for optimizing agent actions. Finally, enabling user input for direct agent control and conducting user studies on perceived agent collaboration would provide valuable insights into human-agent interaction.

References

Tolga Abaci, Jan Ciger, and Daniel Thalmann. 2005. Planning with smart objects. In *International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision*.

Ramiro A Agis, Sebastian Gottifredi, and Alejandro J García. 2020. An event-driven behavior trees extension to facilitate non-player multi-agent coordination in video games. *Expert Systems with Applications* 155 (oct 2020), 113457. doi:10.1016/j.eswa.2020.113457

Stefano V Albrecht, Filippos Christianos, and Lukas Schäfer. 2024. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches.* MIT Press. https://www.marl-book.com

Jasmine Jerry Aloor, Siddharth Nagar Nayak, Sydney Dolan, and Hamsa Balakrishnan. 2024. Cooperation and Fairness in Multi-Agent Reinforcement Learning. *ACM Journal on Autonomous Transportation Systems* 2, 2 (dec 2024), 1–25. doi:10.1145/3702012

Abdollah Amirkhani and Amir Hossein Barshooi. 2022. Consensus in multi-agent systems: a review. *Artificial Intelligence Review* 55, 5 (2022), 3897–3935. doi:10.1007/s10462-021-10097-x

Alex J Champandard and Philip Dunstan. 2019. The behavior tree starter kit. In *Game AI Pro 360: Guide to Architecture*. CRC Press, 27–46.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, and Others. 2023. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *arXiv preprint arXiv:2308.10848* 2, 4 (2023), 6.

Michele Colledanchise, Diogo Almeida, and Petter Ogren. 2019. Towards Blended Reactive Planning and Acting using Behavior Trees. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 8839–8845. doi:10.1109/icra.2019.8794128

Michele Colledanchise, Alejandro Marzinotto, and Petter Ogren. 2014. Performance analysis of stochastic behavior trees. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3265–3272. doi:10.1109/icra.2014.6907328

Michele Colledanchise and Lorenzo Natale. 2018. Improving the Parallel Execution of Behavior Trees. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 7103–7110. doi:10.1109/iros.2018.8593504

Ali Dorri, Salil S Kanhere, and Raja Jurdak. 2018. Multi-Agent Systems: A Survey. *IEEE Access* 6 (2018), 28573–28593. doi:10.1109/access.2018.2831228

Sven Gronauer and Klaus Diepold. 2022. Multi-agent deep reinforcement learning: a survey. *Artificial Intelligence Review* 55, 2 (2022), 895–943. doi:10.1007/s10462-021-09996-w

Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. 2022. A survey of Behavior Trees in robotics and AI. *Robotics and Autonomous Systems* 154 (aug 2022), 104096. doi:10.1016/j.robot.2022.104096

Anja Johansson and Pierangelo Dell'Acqua. 2012. Emotional behavior trees. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 355–362. doi:10.1109/cig.2012.6374177

Huimin Liu, Minsoo Choi, Dominic Kao, and Christos Mousas. 2023. Synthesizing Game Levels for Collaborative Gameplay in a Shared Virtual Environment. *ACM Trans. Interact. Intell. Syst.* 13, 1 (mar 2023). doi:10.1145/3558773

Siqi Liu, Guy Lever, Nicholas Heess, Josh Merel, Saran Tunyasuvunakool, and Thore Graepel. 2019. Emergent Coordination Through Competition. In *International Conference on Learning Representations.* https://openreview.net/forum?id=BkG8sjR5Km

Ryan Lowe, YI WU, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/68a9750337a418a86fe06c1991a1d64c-Paper.pdf

Aadesh Neupane and Michael Goodrich. 2019. Learning Swarm Behaviors using Grammatical Evolution and Behavior Trees. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-2019)*. International Joint Conferences on Artificial Intelligence Organization, 513–520. doi:10.24963/ijcai.2019/73

Renato de Pontes Pereira and Paulo Martins Engel. 2015. A Framework for Constrained and Adaptive Behavior-Based Agents. doi:10.48550/ARXIV.1506.02312

Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. 2016. Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving. doi:10.48550/ARXIV.1610.03295

Xiumin Shang, Tengyu Xu, Ioannis Karamouzas, and Marcelo Kallmann. 2023. Constraint-based multi-agent reinforcement learning for collaborative tasks. *Computer Animation and Virtual Worlds* 34, 3–4 (may 2023). doi:10.1002/cav.2182

Alexander Shoulson, Max L Gilbert, Mubbasir Kapadia, and Norman I Badler. 2013. An Event-Centric Planning Approach for Dynamic Real-Time Narrative. In *Proceedings of Motion on Games (MIG '13)*. ACM, 121–130. doi:10.1145/2522628.2522629

Christopher Iliffe Sprague and Petter Ogren. 2022. Adding Neural Network Controllers to Behavior Trees without Destroying Performance Guarantees. In *2022 IEEE 61st Conference on Decision and Control (CDC)*. IEEE, 3989–3996. doi:10.1109/cdc51059.2022.9992501

Enrique Areyan Viqueira and Cyrus Cousins. 2019. Learning simulation-based games from data. In *Proceeding AAMAS'19 Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, Vol. 2019.

Xiangke Wang, Zhiwen Zeng, and Yirui Cong. 2016. Multi-agent distributed coordination control: Developments and directions via graph viewpoint. *Neurocomputing* 199 (2016), 204–218. doi:10.1016/j.neucom.2016.03.021

G Weiss. 1999. *Multiagent systems: a modern approach to distributed artificial intelligence.* MIT press.

Xun Zhang, Davide Schaumann, Brandon Haworth, Petros Faloutsos, and Mubbasir Kapadia. 2019. Coupling agent motivations and spatial behaviors for authoring multiagent narratives. *Computer Animation and Virtual Worlds* 30, 3–4 (may 2019). doi:10.1002/cav.1898

Yongqi Zhang, Haikun Huang, Erion Plaku, and Lap-Fai Yu. 2021. Joint Computational Design of Workspaces and Workplans. *ACM Transactions on Graphics* 40, 6 (2021).