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

A HIGH EXTENSIBLE MODELING METHOD USING THREE-LAYER COMPONENT-BASED ARCHITECTURE

Haozhe Yuan[1], Yiping Yao[1], Wenjie Tang[1], and Feng Zhu[1]

[1]College of Systems Eng., National University of Defense Technology, Changsha, CHINA

ABSTRACT

Extensibility and reusability are important yet competing objectives in the modeling process. Despite the progress made by current modeling methodologies, they tend to be limited by either one-way control transfers or static data linkages. Considering these limitations, we introduce a novel three-layer, progressive, composable modeling architecture that divides the system model into three layers: components, entities, and systems. It incorporates behavior trees for the assembly of functional components into entities and adopts a publish-subscribe communication paradigm to dynamically establish interactions among entities. Case studies confirmed that this approach facilitates the efficient development of simulation models. Moreover, it allows for the agile adaptation of entity behavior models and their interconnections, ensuring robust extensibility and optimizing the reuse of models as simulation requirements evolve.

1 INTRODUCTION

Computer simulations provide a repeatable experimental environment to study various systems, allowing researchers to observe system behavior under different conditions for diverse research purposes. In practice, researchers frequently adjust and refine models in response to changes in the research objectives, necessitating a model architecture with high extensibility. Alternatively, the reuse of validated models has garnered interest to enhance modeling efficiency and ensure simulation reliability. However, these two goals are conflicting and present significant challenges. The component-based modeling approach has demonstrated potential to enhance model extensibility, reusability, and flexibility.

A kind of component-based modeling strategies represent models as several components interconnected through ports to forge a complete model. The primary challenge lies in the need for ports to be precisely defined using hard coding, leading to static data linkages within the system and restricting the decoupling between components, such as the discrete-event system specification (DEVS) framework (Zeigler 1976), which componentizes the model structure through hierarchical recursion; however, the internal port connections of the coupling model are hard-coded, hindering the support of flexible adjustments in the inter-entity interactions. The other kind of strategies separate the model's behavior from its specific logical functions, thus facilitating function modularization. The effectiveness of this strategy depends largely on the behavior modeling method. State machines have been widely adopted to outline behavioral logic. Such a framework often entails one-way control transfers, wherein the control flow is transferred between components, continuing its operation within a new context without reverting to the original point of invocation (Colledanchise and Ogren 2018). This configuration can yield an intricate model structure, increasing the complexity of model maintenance. Modifications to a single component can trigger changes in other parts of the model, limiting its ability to expand flexibly and accommodate new modeling objectives. Static data linkages and one-way control transfers are the main challenges in component-based modeling research.

We propose a hierarchical modeling architecture that integrates the use of the aforementioned two component strategies at different levels and has made improvements to address the issues they each face. The system is organized into a series of entities that interact via discrete events to yield a comprehensive

model. Within an entity, event processing and generation are implemented by invoking functional components. The system model can be assembled in two steps: 1. *Components to Entity*. This phase is based on the idea of separating behavior from functionality and modularizing the functionalities. It models entity behaviors using a behavior tree structure (Iovino et al. 2022) constrained by the postcondition-precondition-action (PPA) paradigm, invoking various functional components to handle events. In the structure of behavior trees, there are dedicated nodes responsible for managing control transfers. The control token is guaranteed to be returned to the parent node after being passed to a child node, forming a two-way control transfer mechanism. This is beneficial for the modularization of behavior models, as adding or removing a node or even a subtree does not affect other nodes, making it easy to expand; 2. *Entities to System*. This subsequent phase employs the publish-subscribe paradigm (Crowley 2008), whereby it establishes data links among entities through real-time matching of publish/subscribe interests, enabling entity assembly into the system model through a dynamic interaction network. This strategy allows a system model to be incrementally constructed from components to entities, and ultimately to a holistic system, thereby introducing two-way control transfers and dynamic interactions among the model's constituents to enhance the reusability and extensibility of the model. This paper presents a case study of a system of firefighting drones for extinguishing small-scale forest fires, confirming that the proposed method supports the effective construction of system models, enables efficient model expansion and existing model reusability as the simulation objectives change.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction and analysis of existing component-based modeling methods. Section 3 presents the background knowledge of behavior trees and the PPA paradigm. Section 4 details the specific content of the three-layer progressive assembly architecture. Section 5 presents case studies and result analysis. Lastly, Section 6 summarizes the findings of this study and proposes future research directions.

2 RELATED RESEARCH

Many useful methodologies exist for the direct decomposition and reconfiguration of model structures, such as the event graph (EG) method (Schruben 1983), a discrete-event simulation (DES) technique that elucidates discrete-event systems by mapping events and their interrelations. Nonetheless, the specification of event interactions, a task undertaken during the design and development stages, poses challenges in terms of adaptability to modifications of interaction dynamics. Innovations such as listener EG objects (Buss and Blais 2007) and object-oriented EGs (Tiacci 2020) integrate object-oriented principles with EGs, elevating the modularity of EG models. However, these methods do not refine the static interactions among objects. System models based on Petri nets (Murata 1989), which are structurally clear and well-integrated. Despite their structural merits, Petri nets exhibit tight coupling that conflicts with the demand for flexible model adjustments. Amparore et al. (2024) proposed the segmentation of large systems into components developed in Petri net formalism. This method uses labels instead of component IDs to identify port interactions, which significantly enhances the degree of inter-component loose coupling. However, it does not fully address the challenges posed by the one-way control transfer characteristics of Petri nets, which have a structure similar to the state machine. The DEVS framework proposed by Zeigler for specifying discrete-event systems supports hierarchical model development by using a coupled model as a fundamental component in another coupled model (Palaniappan et al. 2006). However, the hierarchical assembly of DEVS-based system models is implemented using input/output ports and message couplings, which means that all pairwise data links between components must be explicitly defined during the system design phase. Thus, the complexity will increase when the system developer adds a new layer to the system, which markedly limits system model extensibility. For the sake of mathematical rigor, DEVS disperses the behavior of components into different functions but is often limited by the readability factor during the modeling and programming process. Architecture analysis and design language (AADL)-DEVS (Ahmad and Sarjoughian 2023) leverages the complementary advantages of the two modeling standards at different design levels, but AADL and DEVS both define the connections between components through hard coding and do not address problems with static data linkages. The network simulator J-Sim (Sobeih et al. 2007),

which is based on the autonomous component architecture (ACA), shares similarities with DEVS but enhances inter-component decoupling. Two components, acting as the initiator (caller) and reactor (callee), are bound at the system integration time to fulfill the contract. From the perspective of design patterns, this study applies the mediator pattern. However, the existence of the mediator contract class means that, when the system needs to be expanded, the mediator complexity and management difficulty will increase.

Another strategy in component-based modeling involves separating a model's behavior from its specific functions, thereby facilitating functional componentization. MAXSIM, a combat simulation platform, features a behavior editor predicated on state machines, allowing users to visually modify model behaviors. When modeling a complex system, multiple one-way control transfers exist. The basic object model (BOM) (Gustavson 2001) was developed to address the challenges of poor reusability and redundancy associated with federation object models within high-level architectures. Despite this, the BOM continues to rely on state machines to model transitions between steps in pattern description tables. This approach inevitably results in complex control transfers and dependencies. In the three-layer architecture for complex adaptive systems (CAS) (Zhu et al. 2017), one or more CAS models are built from lower-level components. The bottom layer is a simulation model component (SMC) that implements some rule-specific support functionality. The middle layer is a logical process (LP) model that states that an agent can react to a current situation by executing a sequence of SMCs. The top layer is the CAS system model, which defines a CAS model consisting of several LPs and their interactions. Ding et al. (2023) developed a flexible operation simulation platform that integrates the three levels of campaign systems, tactical entities, and technical components. Fine-grained electromagnetic models are included in large-scale system simulations to create highly realistic electromagnetic environments. However, for both of the two methods, the data linkages between entities are fixed and the entity behavior models are rigid, rendering them inflexible to changes in the research objectives.

The aforementioned methods effectively facilitate the component-based construction of simulation models. However, they are constrained by the one-way control transfer-related complexity of the behavior model or static data linkage-related difficulty in extending component interactions. To address these issues, we drew upon the concept of assembly at the system, logical entity, and technical component levels and replaced hard-coded interactions with a publish-subscribe paradigm to achieve the inter-entity dynamic matching of data connections； further modularized the functionality of entities, and organized them into a behavior tree structure to enable two-way control transfer and enhance the readability of the model code.

3 BACKGROUND

3.1 Behavior Tree

Behavior trees originated in the computer gaming industry and are widely used in robotics and games. Behavior trees can be formally defined as $BT = < V, E, \tau >$, where V denotes the set of tree nodes, E represents the set of edges, and $\tau \in V$ denotes the root node. Execution within a behavior tree begins at the root node, disseminating a tick signal downstream. Each node responds to this signal by performing designated functions and returns one of three statuses: SUCCESS, FAILURE, or RUNNING. The nodes in the tree are categorized into two main types: control nodes (non-leaf nodes), which govern the propagation of tick signals among child nodes, and execute nodes (leaf nodes), which execute specific condition checks or actions. Classic control nodes include sequence nodes that execute child nodes in order until one returns a FAILURE status or all return SUCCESS and selector nodes that execute child nodes sequentially until one returns SUCCESS or all return FAILURE.

Diverging from finite state machines, behavior trees maintain a call-and-return dynamic between nodes, whereas the reactivity feature ensures the correct execution sequence. A tree structure, rather than a network structure, has the advantages of modularity and readability, which allow developers to tailor behavior trees to meet specific modeling requirements.

3.2 PPA Paradigm

The PPA paradigm is a design pattern for behavior trees. The postcondition represents the objective of the behavior, the precondition denotes the conditions that must be met to achieve the goal, and the action signifies the operation required to fulfill the postcondition based on satisfaction of the precondition. These three elements are shown in the tree structure of Figure 1. For instance, when entering a building is the goal, the precondition would be an open door and the action would be entering through the door. Additionally, a precondition can serve as a postcondition for another PPA subtree, further decomposing it.

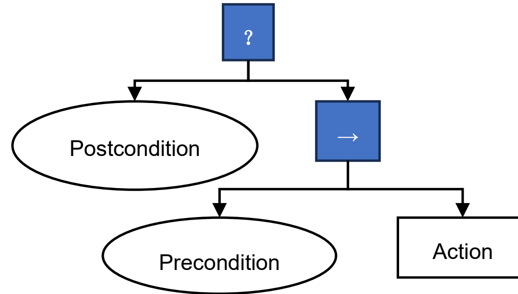


Figure 1: PPA subtree structure.

4 MODELING ARCHITECTURE

Our architecture divides the system model into three layers: components, entities, and systems. Initially, in the *Components to Entity* phase, behavior trees are utilized to integrate functional components. Subsequently, in the *Entities to System* phase, as the entity model specifies the types of data allowed for publishing and subscribing, the simulator aligns these publish and subscribe interests to dynamically forge data linkages among entities, thereby constructing the system model. This ultimately forms a progressive *Component to Entity to System* assembly modeling architecture.

4.1 Formalism of the Entity Model

Embracing the discrete-event system perspective, we modeled a simulation entity using the following octet:

$$Entity =< S_{int}, I_{env}, X, Y, T_{sub}, T_{pub}, f_{init}, f_{bt} >$$

Here, S_{int} represents the set of internal states derived from the aggregation of the entity's internal attributes: $S_{int} = \times_{i=1}^{m} attri_i$.

I_{env} denotes the environmental information set, or external state set, representing the dataset of external environmental mappings perceived by the entity, as derived from the environmental attributes of interest to the entity: $I_{env} = \times_{i=1}^{n} env_i$.

X and Y denote the input and output sets, respectively. In discrete-event systems, an entity's response to an input event involves executing an action, which alters its state, or scheduling an output event. Defined within the DES paradigm, an event comprises two components: a timestamp (time of occurrence) and event content (custom structure): $X = \{< timestamp, content >\}$, $Y = \{< timestamp, content >\}$.

T_{sub} and T_{pub} represent the sets of interests subscribed to and published by the entity, i.e., the entity can send and receive messages on the corresponding topic data: $T_{sub} = \{topic\}$, $T_{pub} = \{topic\}$. Under the publish-subscribe paradigm, a $topic$ acts as a message-routing identifier, which can be of any data category, tailored by the modeler to fit the modeling goals and system characteristics. In addition to a simple string form, some systems permit the adoption of complex topics with attributes, allowing subscribers to apply fine-grained filtering.

f_{init} denotes the initialization function. This function generates the entity's initial state and a set of output events, initiating the simulation based on predefined experimental scenarios: $f_{init}: scenario \rightarrow S_{int} \times Y \cup \{\emptyset\}$.

f_{bt} represents the behavior function. The logical functions of entities, such as motion and communication, were designed as different functional components incorporated within the behavior function to introduce two-way control transfers and refine the design constraints for model behavior. It updates the internal states, environment information, and publish/subscribe interests based on the internal states, environment information, and new inputs: $f_{bt}: S_{int} \times I_{env} \times X \rightarrow S_{int} \times I_{env} \times Y \cup \{\emptyset\} \times T_{sub} \times T_{pub}$.

Figure 2 shows the architecture of the entity model. The function call reflects the relationship between the entity and component models, whereas the read/write operation entails f_{bt} and f_{init} reading data from or writing data to other elements of the entity model.

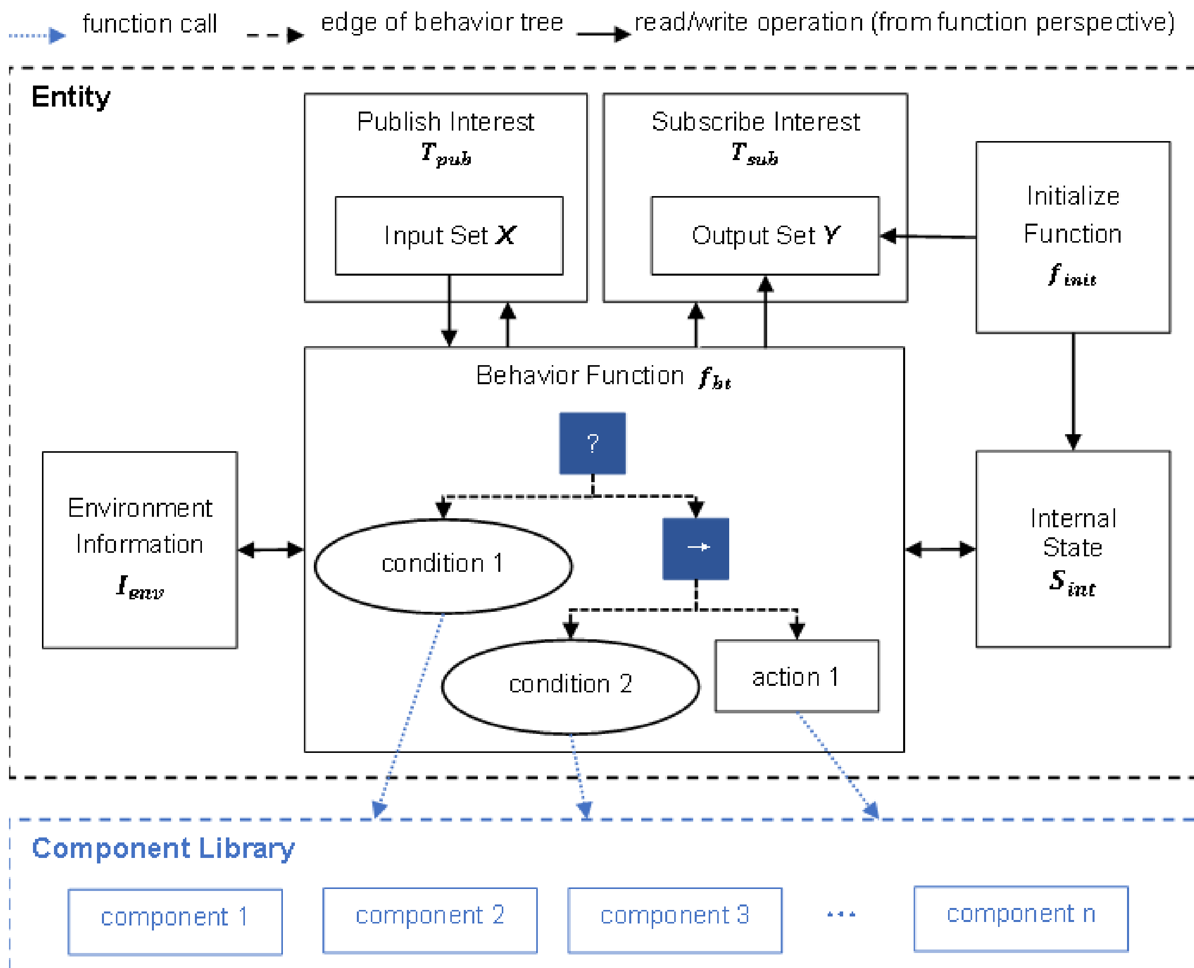


Figure 2: Entity model architecture.

4.2 *Components to Entity*: Behavior Function Assembles Components

This method separates behavior from functionality at the entity level and uses a behavior tree to assemble functional components into an entity's behavior function.

By constructing behavior trees based on the PPA paradigm, design constraints were introduced into the entity behavior modeling process, further optimizing the model structure. As shown in Figure 1, within the modeling phase, the entity behavior objectives (postcondition), actions required to achieve these objectives (action), and prerequisite conditions (precondition) are placed in the corresponding condition and action nodes of the PPA subtree structure. Each node, condition, or action invokes the corresponding functional component to address the specific logic it represents. Through the recursive augmentation of PPA subtrees, developers progressively refine the behavior function of the entity model until further decomposition of the precondition nodes is untenable.

Fundamentally, the entity's behavior function manifests as a behavior tree formulated via the PPA paradigm, defined as $f_{bt} =< V, E, \preccurlyeq >$. The node set $V = \{< type, func >\}$. Here, $type =< control, execute >$. Control nodes' $func$ elements dictate tick signal propagation based on predetermined logic, whereas the $func$ of execute nodes expresses a specific function of the entity behavior. The $func$ of execute nodes reads data from the entity's internal state set, environmental information set, and input event content, and calls functional components for computation.

The execute nodes are further divided into two categories: conditions and actions. Condition nodes within the behavior tree activate sensor-type functional components, assimilate data from input events, combine them with internal state and external environmental data for condition judgment, and subsequently store outcomes in I_{env} for access by action nodes. To foster component reusability while acknowledging that various condition nodes may require identical functional components, condition nodes were designed to follow functional component output with additional evaluative logic. For instance, within a behavior tree incorporating both ally and adversary distance evaluative nodes, a single sensor-type functional component, class-TargetDistance, is sufficient to interpret the position data from target entities. These condition nodes independently assess the affiliation and proximity of the target entity. Alternatively, action nodes are more straightforward because they extract the requisite data from internal and environmental sources to facilitate component-executed tasks.

Algorithm 1 describes the operational logic of the behavior function. Each iteration sets the execution pointer at the root node (Line 1), initiating movement and execution across nodes based on their $func$ elements (Line 6, Line 10). Nodes return states, such as SUCCESS, FAILURE, or RUNNING (Line 6, Line 9). This iterative process continues until the root node state of the Behavior Tree is altered (Line 2).

Algorithm 1: Pseudocode for behavior function.

1	$pointer \leftarrow node_0$;
2	**while** not $node_0.state == Invalid$ **do**:
3	$pointer \leftarrow pointer.next$;
4	**switch**($pointer.type$)
5	**case** $control$:
6	$poniter.state, pointer.next \leftarrow func(pointer.child.state, E, \preccurlyeq)$;
7	**break**;
8	**case** $execute$:
9	$pointer.state, S_{int}, I_{env}, y, T_{sub}, T_{pub} \leftarrow func(x.content, S_{int}, I_{env})$;
10	$pointer.next \leftarrow pointer.last$;
11	**break**;
12	**return** S_{int}, I_{env}, y

In summary, the entity model manages the invocation of the component model by using behavior functions, capitalizing on the inherent strengths of behavior trees, including two-way control transfer capability, modularity, and enhanced readability. These attributes facilitate the flexible and precise adjustment of entity behaviors, significantly enhancing model extensibility and reusability.

4.3 *Entities to System*: Dynamic Interest Matching Among Entities

After assembly of the components into the behavior function of an entity, this architecture dynamically matches inter-entity interactions based on their publishing/subscribing interest sets. Here, this task was completed by using a simulation platform, i.e., a simulator. The simulator was equipped with an event list and interest-matching module. Algorithm 2 provides the pseudocode for the simulator.

The event list catalogs the upcoming events. When an entity generates an output (Lines 4–7 and 23–27 in Algorithm 2), the simulator positions it accurately within the list according to its timestamp (Lines 3–9). Next, the simulator dispatches the imminent event to the pertinent entity (Line 13), initializing the entity's behavior function to address the event (Lines 16–18) and adjusting the simulation clock to align with the event's timestamp (Line 14).

Unlike conventional interest-matching algorithms that find the target entities for every event, our methodology implements an entity-to-entity matching mechanism within its interest-matching module to minimize redundant calculations. During the simulation initialization phase (Line 2), the interest-matching module delineates a normalized multidimensional space per topic by assigning each dimension to an attribute under the topic. It then maps all entity publish and subscribe interest sets to a multidimensional space, forming each entity's publish and subscribe regions. The intersecting regions denote potential data-exchange pathways between entities across various topics. The methodology also permits dynamic adjustment of the entity interest sets during simulation, necessitating recalibrations solely for the affected entities (Lines 20–21).

Algorithm 2: Pseudocode for the simulator.

```
1   time ← 0;
2   destmap ← interesr_match(all T_pub, all T_sub);
3   #Handle initial events
4   y_init ← pubentity.f_init(scenario);
5   event.timestamp ← y_init.timestamp;
6   event.destlist ← destmap.find(y_init.topic);
7   event.content ← y_init.content;
8   #Insert initial events into event list of simulator
9   eventlist.insert(event);
10  #Start running
11  while not termination_condition(time) do:
12    #Fetch new event
13    next_event ← eventlist.pop_front();
14    time ← next_event.timestamp;
15    x ← next_event;
16    for destentity in x.destlist;
17      #Trigger the behavior function of destination entity
18      destentity → f_bt(x.content);
19      #Update the results of interest match
20      if there is pubtopic/subtopic change:
21        destmap.update();
22      #Generate a new event and insert it into the simulator's event list
23      else if there is new y:
24        event.timestamp ← y.timestamp;
25        event.destlist ← destentity.destmap.find(y.topic);
26        event.content ← y.content;
27        eventlist.insert(event);
```

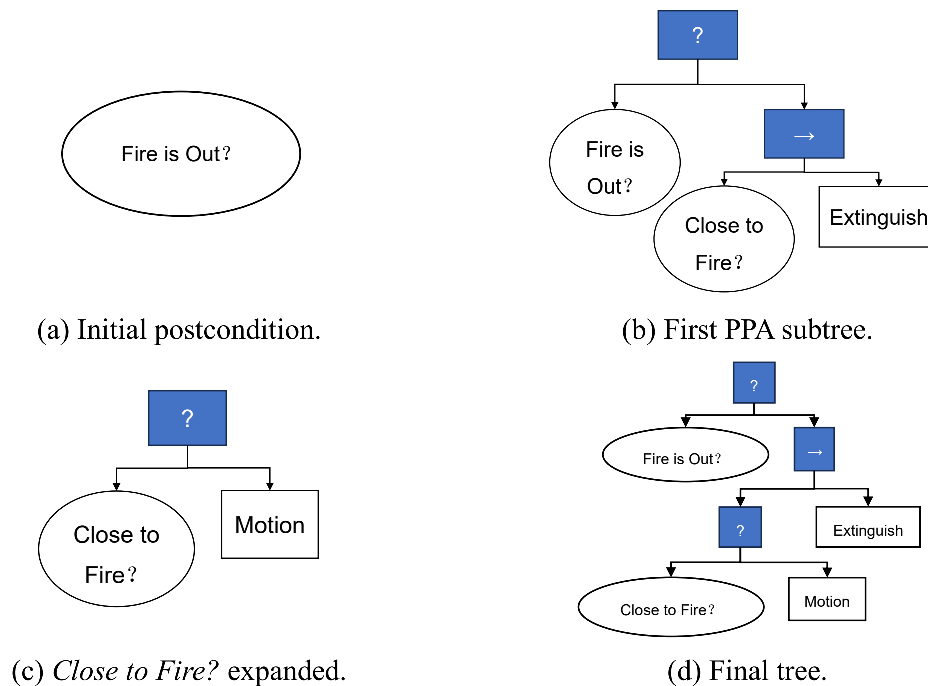
Thus, the simulator can dynamically establish inter-entity communication links with minimal computational overhead, effectively structuring the system model. If model expansion is required, the developer is merely tasked with revising the entity publish and subscribe sets, independent of other entities' presence or modifications, which simplifies and expedites the model augmentation process.

5 CASE STUDY

This case study consisted of two phases. (1) The firefighting drone (*Firefighter*, entity ID 309191) navigates toward the forest fire (*Fire*, entity ID 309201) along a predefined route and deploys extinguishing bombs (*Extinguisher*, entity ID 309191000). The effectiveness of the architecture can be evaluated by analyzing the behavioral transitions of entities and exchange of messages as recorded in the simulation logs. (2) A simulation incorporating mountainous terrain features (*Mountain*, entity ID 309211) and autonomous fire detection and response by the drone is developed. The drone must navigate the terrain to avoid collisions. We will analyze its extensibility and reusability by adjusting the process of entities and system models.

5.1 Phase 1: Model Development and Validation

First, we modeled the behavioral functions of each entity. Using Firefighter as an example, Figure 3 illustrates the PPA paradigm-based three-step construction process for the behavior tree: In Figure 3(b), *Fire is Out?* is the postcondition, *Extinguish* is the action, and *Close to Fire?* is the precondition that serves as the postcondition in the subtree shown in Figure 3(c).



(a) Initial postcondition.

(b) First PPA subtree.

(c) *Close to Fire?* expanded.

(d) Final tree.

Figure 3: Modeling process for Firefighter behavioral function.

The *Fire is Out?* and *Close to Fire?* nodes of Firefighter call the ModelProcess method of the target state recognition component class TargetDamage and target distance judgment component class TargetDistance, respectively, to realize the purpose of the node. Firefighter also calls the extinguish component via the *Extinguish* node to publish an entityCreate message to Extinguisher to activate it. The *Motion* node is included in the behavioral functions of the three entities, all of which call the motion component to move to the navigation point (i.e., change their own position and publish a new posChange

message). Their differences are as follows: the navigation point of Firefighter is given in the scenario, that of Fire is empty, and that of Extinguisher is determined according to the location of the Fire. In addition to having nodes identical to those of Firefighter, Extinguisher has the *Is Launched?* node, which can judge whether it is activated, and the *Detonate* node, which publishes the detonation message to entities within range. In addition to the *Motion* node, Fire also needs to check its own survival status and publish it externally through the node *Is Damaged?*.

The system model structure is shown in Figure 4, which illustrates the attributes of the entities' internal state sets and topics in the publish/subscribe interest sets. The behavioral functions are also presented in the form of behavioral trees.

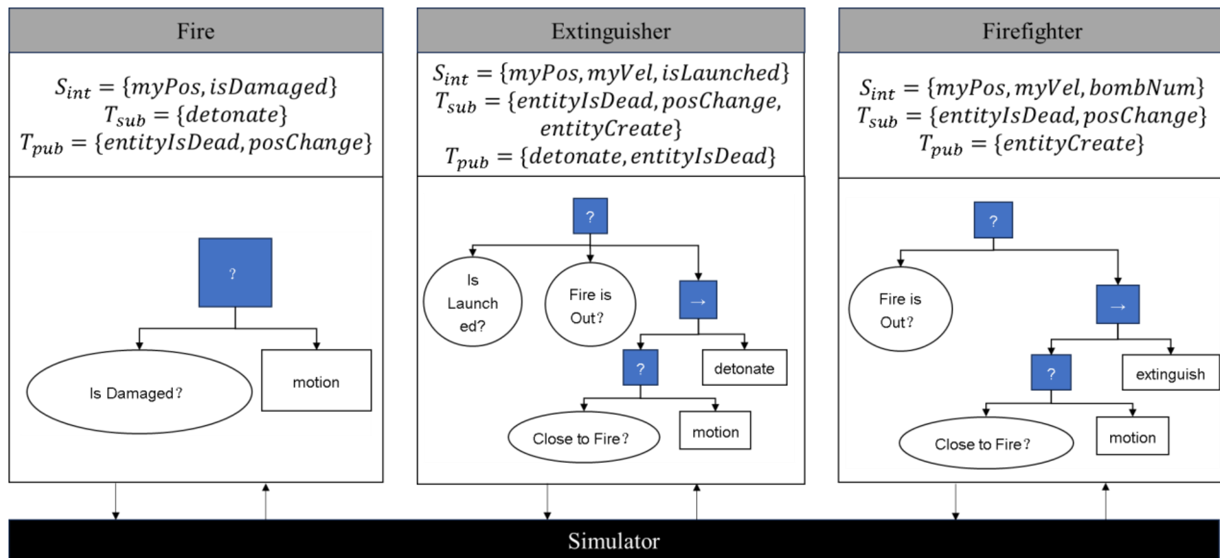


Figure 4: System model structure.

Figure 5 shows a frame of the simulation running process, depicting Firefighter in motion toward Fire at simulation times of 20 and 20.5; a detailed log of entity behaviors is also shown. The entities were designed to dispatch a self-scheduling message at 0.5 intervals; therefore, Firefighter will repeat the behaviors taken at time 20 at time 20.5.

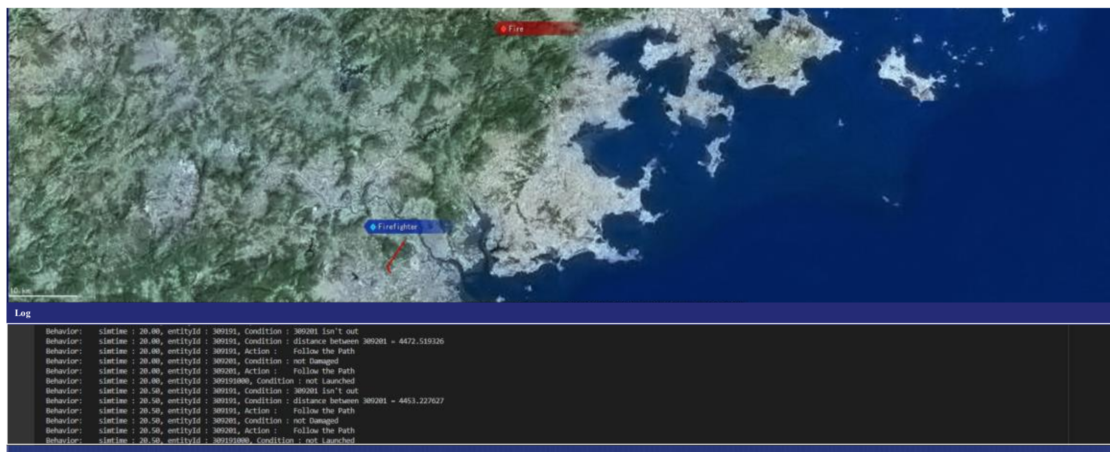


Figure 5: Frame of simulation log.

Table 1 details the nodes activated within the behavioral functions of each entity at 20, along with the outcomes of their execution, referred to as entity-executed actions. The table also lists the topic of transmitted messages associated with these actions.

Table 1: Simulation log for simulation time (Simtime) = 20.00.

Simtime	Entity ID	*Node being ticked:* Behavior	Topic of sent message
20.00	309191	*Fire is Out?:* 309201 is not extinguished	
20.00	309191	*Close to Fire?:* Distance from 309201 = 4472.52	
20.00	309191	*Motion:* Motion	
20.00	309201	*Is Damaged?:* No damage	
20.00	309201	*Motion:* Motion	posChange
20.00	309191000	*Is Launched?:* No launch	

Table 2 highlights a pivotal moment, i.e., upon the *Close to Fire?* node of Firefighter determining that the proximity to Fire falls below the 200-unit threshold, the entity promptly activates the *Extinguish* node. This action sends an entityCreate message, triggering the activation of Extinguisher. Upon reception of the entityCreate message, the *Is Launched?* node's conditional check is satisfied, thereby enabling initiation of the subsequent behavioral sequence.

Table 2: Simulation log for Simtime = 131.00–131.10.

Simtime	Entity ID	*Node being ticked:* Behavior	Topic of sent message
131.00	309191	*Fire is Out?:* 309201 is not extinguished	
131.00	309191	*Close to Fire?:* Distance from 309201 = 189.77	
131.00	309191	*Extinguish:* Extinguish	entityCreate
131.00	309201	*Is Damaged?:* No damage	
131.00	309201	*Motion:* Motion	posChange
131.10	309191000	*Is Launched?:* Launched	
131.10	309191000	*Fire is Out?:* 309201 is not extinguished	
131.10	309191000	*Close to Fire?:* Distance from 309201 = 168.89	
131.10	309191000	*Motion:* Motion	

Lastly, Table 3 details the instant when Extinguisher, upon detecting a distance of less than 20 units from Fire, detonates and subsequently alters Fire's state of existence.

Table 3: Simulation log for Simtime = 134.60–134.70.

Simtime	Entity ID	*Node being ticked:* Behavior	Topic of sent message
134.60	309191000	*Is Launched?:* Launched	
134.60	309191000	*Fire is Out?:* 309201 is not extinguished	
134.60	309191000	*Close to Fire?:* Distance between 309201 = 14.02	
134.60	309191000	*Detonate:* Detonate	detonate, entityIsDead
134.70	309201	*Is Damaged?:* Damaged	entityIsDead
134.70	309191	*Fire is Out?:* 309201 is extinguished	

5.2 Phase 2: Model Extension and Flexibility Analysis

In the subsequent phase, the system introduces Mountain, which also utilizes a *Motion* node with an empty navigation point and publishes events under the posChange topic without subscribing to any events. PosChange events from Mountain were monitored by all entities capable of movement within the system.

Incorporating this new entity merely entails updating the subscribe sets of Firefighter and Extinguisher to reflect the altered interaction dynamics.

Evolution of the simulation model for this phase led to the streamlined creation of new behavioral functions for Firefighter, maintaining the PPA tree structure (Figure 6). The *No collision risk?* node assesses the collision risk based on the analytical outcomes derived from the position data of Mountain by using the TargetDistance component. The subtree enclosed in the red box represents the final behavioral function of Firefighter, which is fully preserved and reused.

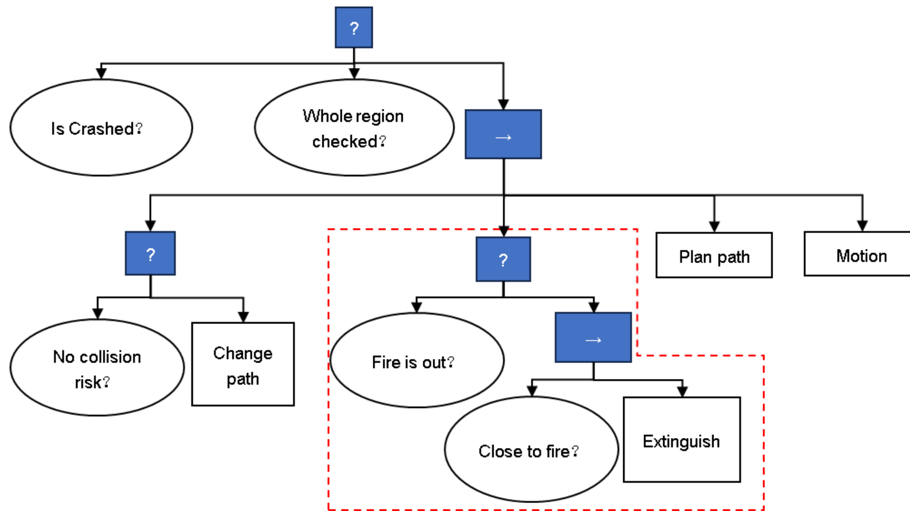


Figure 6: Updated behavioral function of Firefighter.

This model modification process underscored the reusability and extensibility of the proposed architecture, indicating that the simulation model can effectively modify the behavioral logic and interaction networks of entities to align with new simulation goals and fully exploit preexisting components and behavioral functions.

6 CONCLUSION

This study yielded a highly extensible modeling methodology that categorically disaggregates a system into three hierarchical levels: components, entities, and systems. The integration of behavior trees, characterized by their robust two-way control transfers and enhanced readability, enables adaptable modifications to entity models. The implementation of publishing and subscribing interests within entities' input/output configurations supports the establishment of dynamic inter-entity interactions. This three-layer *Component to Entity to System* assembly architecture's efficacy, extensibility, and reusability has been verified through the empirical case study delineated here.

Considering the large amount of work on the automatic generation of behavior trees, the proposed architecture can be combined with artificial intelligence technology to explore the spontaneous organization of entity models to produce the correct behavioral logic. This would further enhance system model extensibility, enabling autonomous learning of and adaptability to more complex and uncertain environments to increase simulation efficiency and result accuracy.

REFERENCES

Ahmad, E. and H. S. Sarjoughian. 2023. “An Environment for Developing Simulatable AADL-DEVS Models”. *Simulation Modelling Practice and Theory* 123:102690.

Amparore, E., M. Beccuti, P. Castagno, S. Pernice, G. Franceschinis, and M. Pennisi. 2024. "From Compositional Petri Net Modeling to Macro and Micro Simulation by Means of Stochastic Simulation and Agent-Based Models". *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 9(1):1-30.

Buss, A. and C. Blais. 2007. "Composability and Component-based Discrete Event Simulation". In *2007 Winter Simulation Conference (WSC)*, 694-702 https://doi.org/10.1109/WSC.2007.4419663.

Crowley, P. 2008. "A Dynamic Publish-Subscribe Network for Distributed Simulation". In 2008 22nd Workshop on Principles of Advanced and Distributed Simulation. June 3rd-6th, Roma, Italy.

Ding, B., F. Mu, Y. Li, Z. Chen, and C. Liu. 2023. "Design of System Combat Simulation Platform for Complex Electromagnetic Environment". *Journal of System Simulation* 35(02):330-338.

Gustavson, P. 2001. "BOM Study Group Final Report". Simulation Interoperability Standards Organization.

Iovino, M., E. Scukins, J. Styrud, P. Ogren, and C. Smith. 2022. "A Survey of Behavior Trees in Robotics and AI". *Robotics and Autonomous Systems* 154:104096.

Murata, T. 1989. "Petri Nets: Properties, Analysis and Applications". *Proceedings of the IEEE* 77(4):541-580.

Colledanchise, M. and P. Ogren. 2018. *Behavior Trees in Robotics and AI: An Introduction*. Boca Raton: Chapman & Hall/CRC Press.

Palaniappan, S., A. Sawhney, and H. S. Sarjoughian. 2006. "Application of the DEVS Framework in Construction Simulation". In *2006 Winter Simulation Conference (WSC)*, 2077-2086 https://doi.org/10.1109/WSC.2006.322996.

Schruben, L. 1983. "Simulation Modeling with Event Graphs". *Communications of the ACM* 26(11):957-963.

Sobeih, A., M. Viswanathan, D. Marinov, and J. C. Hou. 2007. "J-Sim: An Integrated Environment for Simulation and Model Checking of Network Protocols". In *2007 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, March 26th-30th, Long Beach, USA.

Tiacci, L. 2020. "Object-oriented Event-graph Modeling Formalism to Simulate Manufacturing Systems in the Industry 4.0 Era". *Simulation Modelling Practice and Theory* 99:102027.

Zeigler, B. P. 1976. *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*. New York: Wiley.

Zhu, F., Y. Yao, W. Tang, and J. Tang. 2017. "A Hierarchical Composite Framework of Parallel Discrete Event Simulation for Modelling Complex Adaptive Systems". *Simulation Modelling Practice and Theory* 77:141-156.

AUTHOR BIOGRAPHIES

HAOZHE YUAN received a B.S. degree in Information Management and Information Systems from Harbin Institute of Technology, China, in 2022. He is currently pursuing his M.S. degree in Management Science and Engineering from the College of Systems Engineering, National University of Defense Technology, China. His research interests include modeling and simulation of complex systems and behavior modeling. His email address is yuanhaozhe1023@nudt.edu.cn.

YIPING YAO is a Professor in the College of Systems Engineering, National University of Defense Technology, China. His research interests include high-performance simulation systems, parallel and distributed simulations, cloud computing, and parallel algorithms. His email address is ypyao@nudt.edu.cn.

WENJIE TANG is the corresponding author and an Associate Professor in the College of Systems Engineering, National University of Defense Technology, China. His research interests include high-performance simulation systems, parallel and distributed simulations, and the modeling and simulation of complex systems. His email address is tangwenjie@nudt.edu.cn.

FENG ZHU is an Associate Professor in the College of Systems Engineering, National University of Defense Technology, China. He was an Academic Visitor to the Department of Computing, Imperial College London, U.K., in 2015. His research interests include high-performance computing and the modeling and simulation of complex systems. His e-mail address is zhufeng@nudt.edu.cn.