Expert Systems With Applications 155 (2020) 113457

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

An event-driven behavior trees extension to facilitate non-player multi-agent coordination in video games

Ramiro A. Agis*, Sebastian Gottifredi, Alejandro J. García

Institute for Computer Science and Engineering (UNS-CONICET), Department of Computer Science and Engineering, Universidad Nacional del Sur, Bahía Blanca, Argentina

ARTICLE INFO

Article history:
Received 15 July 2019
Revised 11 February 2020
Accepted 13 April 2020
Available online 23 April 2020

Keywords:
Agents
Coordination
Event-driven behavior trees
Multi-agent systems
Video game development

ABSTRACT

In this paper, we extend *behavior trees* (BTs), a behavior creation method that is popular in the video game industry, with three new types of nodes that facilitate the design and implementation of *non-player characters* (NPCs) that need to coordinate with each other. We provide an implementation and a methodology to use the coordination nodes of our extension appropriately, and we show how to use them to develop an application scenario. In the last years, coordination in multi-agent systems has been a very active research field, both from theoretical and practical points of view. Something similar has happened with the development of new tools for the video game industry. Our approach contributes to both areas by providing a novel extension that facilitates the design and implementation of agents that need to coordinate with each other. In video games, agents or NPCs are—as their name implies—characters that are not controlled by the player but by the game through an algorithmic, predetermined, or responsive behavior, or a more sophisticated AI technique. Some video games require NPCs with dynamic, credible, and intelligently unpredictable behaviors to keep players engaged and immersed. Instead of endowing NPCs with very complex individual behaviors, a feasible way to improve their unpredictability in an intelligent and credible manner is allowing them to coordinate with each other. Since BTs focus on the creation of individual behaviors, coordinated behaviors nowadays tend to be achieved by hard-coding the coordination itself. However, that ad hoc solution partially drives away some of the benefits that popularized BTs: Being visually intuitive, scalable, and reusable. For this reason, we propose an extension to BTs that developers can use to coordinate NPCs without going against the development paradigm: creating complex behaviors by designing an intuitive tree structure.

1. Introduction

Coordination in multi-agent systems is a very active research field that provides theoretical and practical tools for many other areas, see for instance (Asl, Bentahar, Mizouni, Khosravifar, & Otrok, 2014; Hu & Leung, 2017; Proskurnikov & Cao, 2017; Sakurama & Ahn, 2020; Wahab, Bentahar, Otrok, & Mourad, 2016; Wang, Zeng, & Cong, 2016; Zhang & Su, 2019; Zhao, Li, & Zhang, 2017; Zhao & Zhang, 2019; Zou, Su, Li, Niu, & Li, 2019). In this paper, we present and implement a novel approach for agent coordination in multi-agent systems consisting of an extension to *behavior trees* (BTs), a behavior creation method that is popular in the video game industry. The importance of our proposal is that it contributes to these areas by providing a novel extension that facilitates the design and implementation of agents that need to coordinate with each other.

For years, generating interesting and lifelike agents or *non-player characters* (NPCs) has arguably been one of the focuses of AI in the video game industry (Yannakakis, 2012; Yannakakis & Togelius, 2018). NPCs are—as their name implies—characters that are not controlled by the player but by the game through an algorithmic, predetermined, or responsive behavior, or a more sophisticated AI technique. While some video games only rely on NPCs with scripted or trivial behaviors, others require NPCs with dynamic, credible and intelligently unpredictable behaviors to keep the player engaged and immersed in the gameplay (Yannakakis & Togelius, 2018).

In most games, NPCs have a completely individual behavior since they act considering only their current state and/or the player's current state. Instead of endowing NPCs with very complex individual behaviors, a feasible way to improve their unpredictability in an intelligent and credible manner is allowing them

* Corresponding author.
E-mail addresses: ramiro.agis@cs.uns.edu.ar (R.A. Agis), sg@cs.uns.edu.ar (S. Gottifredi), ajg@cs.uns.edu.ar (A.J. García).

https://doi.org/10.1016/j.eswa.2020.113457
0957-4174/

to coordinate with each other. Clearly, whenever a player has an encounter with many NPCs, the higher the amount of possible interactions between the NPCs, the lower the chance that the player predicts their actions.

Some video games, like the horror first-person shooter *F.E.A.R.* (Warner Bros. Interactive Entertainment, 2005), deceive the player into thinking that there are actual coordinated interactions between the NPCs. F.E.A.R.'s AI has always been one of the most acclaimed in the genre given that players claim to be intelligently flanked and ambushed by enemies (soldiers) in unique and irreproducible ways in different playthroughs. Although soldiers in F.E.A.R. do not actually interact with each other and just follow orders from a global AI, the game achieves that false sense of coordination by constantly observing the current state and reproducing appropriate dialog sequences (Orkin, 2006).

Creating an illusion of coordination between NPCs is effective since what matters, after all, is what the players perceive. However, relying on these tricks may not be feasible depending on the game's theme. For instance, F.E.A.R. achieves that effect by making most enemies move in troops and communicate in a language that the player can interpret, which is a characteristic that does not fit with every possible game's theme. This is what motivated this work: Extending a behavior creation method that is popular in the video game industry with a module that facilitates the creation of NPCs with actual coordinated behaviors.

As of today, the video game industry does not take full advantage of the academic research that has been conducted on behavior creation for NPCs (Lemaitre, Lourdeaux, & Chopinaud, 2015; Yannakakis & Togelius, 2018). The main reason is that academic solutions in the field tend to be difficult to reuse or customize, or tend to be unrealistic for the complex software architecture of a complete video game. This has caused behavior creation methods like *behavior trees* (BTs) to dominate the control of NPCs in the video game industry.

BTs became popular for their development paradigm: being able to create a complex behavior by only programming the NPC's actions and then designing a tree structure—usually through *drag and drop*—whose leaf nodes are actions and whose inner nodes determine the NPC's decision making. BTs are visually intuitive and easy to design, test, and debug, and provide more modularity, scalability, and reusability than other behavior creation methods like *finite state machines*. In particular, BTs became popular over a decade ago mainly after their successful application in commercial video games such as *Halo 2* (Isla, 2005; Microsoft Game Studios, 2004), *Halo 3* (Isla, 2008; Microsoft Game Studios, 2007), *Bioshock* (2K Games, 2007), *Spore* (Champandard & Dunstan, 2012; Electronic Arts, 2008), among others. Some examples of recent commercial titles that have confirmed using BTs in their development are *Far Cry: Primal* (Ubisoft, 2016a), *Tom Clancy's The Division* (Ubisoft, 2016b) and *Just Cause 4* (Square Enix, 2018).

Following their popularity in the industry, BTs also started to receive attention in academic research. The authors of (Johansson & Dell'Acqua, 2012) presented a new type of node for BTs that uses the NPC's emotions for decision making. In particular, this node takes into account three factors (time-discounting, risk perception, and planning) to change the execution priority of its children. In (Shoulson, Garcia, Jones, Mead, & Badler, 2011), a method to improve the flexibility of parameter passing in BTs was proposed. In (Lim, Baumgarten, & Colton, 2010), an iterative learning process was used to evolve different BTs to develop an AI-controlled player for the commercial real-time strategy game *DEFCON*. In (Flórez-Puga, Gomez-Martin, Gomez-Martin, Díaz-Agudo, & González-Calero, 2009) the authors apply case-based reasoning techniques to retrieve and reuse stored BTs to dynamically build an NPC's BT at runtime by taking into account the world state and goals. In (Colledanchise, Parasuraman, & Ogren, 2018), the authors propose a model-free automated planner framework using genetic programming that can generate an optimal BT for an autonomous agent to achieve a goal in a fully observable environment. Research on BTs has been relevant not only for video game AI but also in other fields like robotics (Colledanchise & Ögren, 2014; Marzinotto, Colledanchise, Smith, & Ögren, 2014), multi-mission UAV control (Ogren, 2012), semi-autonomous surgery (Hu, Gong, Hannaford, & Seibel, 2015), among others.

Over the years, the diverse implementations of BTs kept improving both in efficiency and capabilities in order to satisfy the demands of the industry (Champandard & Dunstan, 2012) until they evolved into *event-driven behavior trees* (EDBTs). EDBTs solved some scalability issues of classical BTs by changing the way in which the tree internally handles its execution and by introducing a new type of node that can react to events and abort running nodes. Nowadays the concept of EDBT is a standard (even though they are still called "behavior trees" for simplicity) and even the two most popular game development engines provide EDBT implementations. In particular, *Unreal Engine 4*[1] was released including an official EDBT module while *Unity*[2] has many third party modules that can be downloaded from the store.

By taking advantage of its event-drivenness, the contribution of this work consists in extending EDBTs with three new types of nodes, called *coordination nodes*, which facilitate the design and implementation of NPCs that can coordinate with each other through a request protocol. These nodes do not provide more "expressive power" in terms of the coordinated behaviors that could be created by using regular EDBTs. Nevertheless, given that EDBTs (and BTs) focus on the creation of individual behaviors, nowadays coordinated behaviors tend to be achieved by hard-coding the coordination itself in a obscure way. For instance, one of the most common techniques to coordinate multiple NPCs is to put a node in each EDBT that repeatedly calls a hard-coded ad hoc procedure that solves the coordination problem by using a shared structure that can be concurrently accessed and modified. However, not only this kind of solution goes against the development paradigm of behavior trees (*i.e.*, programming only the NPCs' actions and designing a tree structure that determines its decision making) but also partially drives away some of the benefits that popularized them: being visually intuitive, scalable and reusable. For all these reasons, it is necessary a solution to create coordinated behaviors that follows the development paradigm of behavior trees, like the one we propose in this paper.

This paper is organized as follows. In Section 2, we introduce the necessary background on EDBTs. Next, in Section 3, we present coordination nodes together with the intuition behind their use. Section 4 follows with a methodology to use these nodes appropriately. In Section 5, we include an application example. Then, in Section 6, we present the main algorithms of our proposal together with some implementation details and a time complexity analysis. In Section 7, we compare our proposal with other related approaches and then discuss on the benefits of using coordination nodes in comparison to the usual technique of hard-coding coordinated behaviors inside methods. Finally, in Section 8, we present conclusions and comment on future work.

2. Event-driven behavior trees

Over the years, BTs evolved into EDBTs to solve some scalability issues and satisfy the demands from the industry. Commercial video games started to require NPCs with more complex behaviors, which implied the need for larger and deeper trees. Broadly speaking, the scalability issues were solved by introducing a new type

[1] Unreal Engine - https://www.unrealengine.com.
[2] Unity - https://unity.com/.

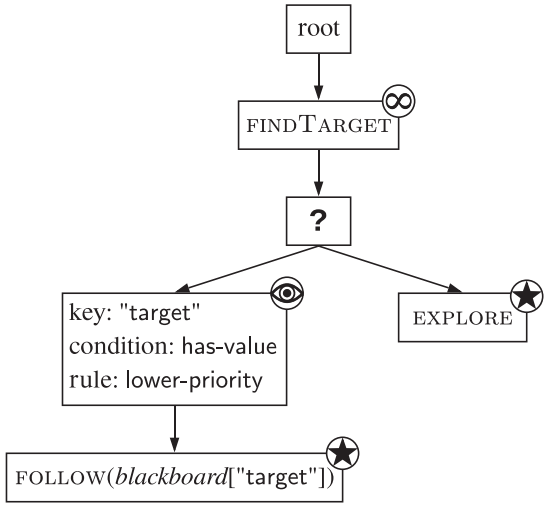

Fig. 1. Event-Driven Behavior Tree: Nodes are represented by rectangles. Nodes' type is depicted with a symbol either inside the rectangle or in the top-right corner. Service nodes are represented by an infinity symbol, selector nodes by a question mark, **BOD** nodes by an eye, and task nodes by a star.

of node that can react to events and abort running nodes and by changing how the trees internally handle their execution: traversing the trees from the root every frame, which is computationally expensive, could be avoided by using a scheduler that stores and updates previously active behaviors. This allowed the design of larger and deeper trees without the performance issues of classical BTs. We refer the interested reader to (Champandard & Dunstan, 2012) for a more detailed explanation about the implementation differences between BTs and EDBTs. In this section, the necessary background on EDBTs is presented. The reader should note that since there are no implementation standards, some of the following concepts or names may differ slightly from other proposed implementations.

2.1. Execution process

An NPC's behavior is determined by the execution of its EDBT, represented by a directed tree (see Fig. 1). The tree's leaves represent all the NPC's possible actions while the inner nodes determine the NPC's decision making. The execution starts by *ticking* its root, and this ticking process traverses downwards through the EDBT's nodes. This process is carried out depending on the ticking rules associated with the type of each node that is ticked.

Each leaf is a *task* node consisting of an NPC's action defined by the EDBT designer (e.g., wander, attack, follow a target, wait, etc.). Whenever a task node is ticked, it enters the running status, and the corresponding action is executed. Then, when the action finishes, the node leaves the running status and returns the *completion status* success or failure to its parent (depending on the action's outcome). For instance, a task node that executes the action WALK may return failure if no path is found to the destination.

On the contrary, inner nodes are divided into *composite* nodes, *service* nodes, and *decorator* nodes, all of which determine the tree's execution flow (*i.e.*, the agent's decision making) by ticking its children according to the node type's rules. Inner nodes return success or failure, depending on their children's completion status or external conditions. The root is a unique node whose only purpose is ticking its only child and stopping the EDBT's execution when it returns a completion status. However, in most popular implementations, the root automatically re-ticks its child whenever the EDBT's execution finishes.

2.2. Composite nodes

Composite nodes can have multiple children and are divided into *selector* nodes, *sequencer* nodes, and *parallel* nodes. Selector nodes tick their children from left to right, one at a time, until one of them returns success or until there are no more children to tick. If a child returns success, the selector returns success; otherwise, it returns failure. Sequencer nodes tick their children from left to right, one at a time, until one of them returns failure or until there are no more children to tick. If a child returns failure, the sequencer returns failure; otherwise, it returns success. Parallel nodes tick all their children simultaneously, allowing multiple subtrees to be executed concurrently. In other words, parallel nodes allow multiple task nodes to be in the running status. Generally, the EDBT designer can customize whether all the children are aborted as soon as one of them returns its completion status, and customize how the parallel node's completion status is affected by its children's. Given that there is no standard, the semantics of this type of node is tied to its implementation. Nevertheless, our proposal does not require any particular type of parallel node, so any desired implementation can be used.

2.3. Service nodes and blackboards

Service nodes have a single composite as a child and are customized with a method and a frequency. Whenever a service node is ticked, it ticks its child and repeatedly calls the method at the specified frequency as long as at least one of the composite's descendants is a task node in the running status. Then, once its child returns a completion status, the service node returns it to its parent. Since the method called by a service node is executed concurrently, nodes in the running status are not interrupted. For this reason, this type of node is often used to make checks and update the EDBT's *blackboard*.

A blackboard is a data structure composed of a dictionary, *i.e.*, a collection of (*key, value*) pairs in which each *key* cannot be associated with more than one *value*. Every EDBT has a private[3] blackboard where all the data that needs to be referenced by its nodes can be stored as (*key, value*) pairs. Although it is not mandatory, in practice the blackboard keys are usually strings. For the rest of the paper, the *value* associated with a *key* in a *blackboard* will be denoted *blackboard*[*key*]. For example, a service node could repeatedly call a method FINDTARGET that checks if there is a target near the NPC and, if any is found, updates *blackboard*["target"] with a reference to it. Then, a task node could fetch this reference from the blackboard and make the NPC follow the target.

2.4. Decorator nodes

Decorator nodes have a single composite or task node as a child. Some examples of decorator nodes are the *conditional decorator*, whose condition determines whether its child is ticked, the *conditional loop decorator*, which works as a conditional decorator but re-ticks its child after it returns a completion status if the condition is still met, and the *loop decorator*, which re-ticks its child after it returns a completion status (a set amount of times or infinitely). A subtype is the *observer decorators*, which was born along with EDBTs for its ability to react to events and abort nodes that are in the running status.

In our proposal, we are particularly interested in the *blackboard observer decorator* (**BOD**). This type of node is composed of a blackboard key, a condition involving that key, and an *abort rule*. Whenever a **BOD** is ticked, if the condition is met, its child is ticked

[3] A discussion on using shared blackboards is included in Section 7.

and its completion status is later returned; otherwise, the **BOD** returns failure. The condition's result and the abort rule both determine whether the **BOD** is registered as an *observer* for the blackboard key in question. Besides the dictionary, blackboards contain a collection of pairs (*key, bod*) called *observers list*, which is updated whenever a *bod* starts or stops observing the *key* it is associated with. Whenever the *value* associated with *key* is added, modified or deleted, if there is a pair (*key, bod*) in the observers list, *bod* is notified by the blackboard and its condition is reevaluated. Then, depending on the location of the node(s) in the running status, the condition's result, and *bod*'s abort rule, the node(s) in the running status may be aborted and *bod* may be ticked, changing the EDBT's execution flow. An abort rule that is important for the rest of the paper is lower-priority, which is based on the concept of lower priority nodes. Given a **BOD** *b*, a node *n* is a lower priority node (with respect to *b*) if *n* is a descendant of *b*'s first composite ancestor that is to *b*'s right. For instance, in Fig. 1 the task node EXPLORE is the only lower priority node for the depicted **BOD**. This section concludes with an example of an EDBT where the abort rule lower-priority is explained.

Example 1. Consider the EDBT depicted in Fig. 1, which models a simple behavior that makes the NPC explore the terrain until it finds a nearby target to follow. Suppose that the blackboard is initially empty and that there are no targets near the NPC. The EDBT's execution starts by ticking its root, which immediately ticks the service node below. This node repeatedly and concurrently calls the method FINDTARGET, which checks if there is a target near the NPC and (if any is found) updates *blackboard*["target"] with a reference to the target. Then, the service node ticks the selector and the selector ticks the **BOD**, which is composed of the blackboard key "target", a condition that checks if *blackboard*["target"] has a value, and the abort rule lower-priority. Given that *blackboard*["target"] has no value, the condition is not met and the completion status failure is returned to the selector. However, the abort rule lower-priority makes the **BOD** start observing the key "target".

The selector then ticks the task node EXPLORE. Suppose that, while the NPC is exploring, FINDTARGET stores a reference to a target in *blackboard*["target"], causing the **BOD** to be notified by the blackboard. Now that the node's condition is met, the abort rule lower-priority has two effects:

1. the **BOD** stops observing the blackboard key "target", and
2. the **BOD**'s first composite ancestor—in this case, the selector—will check if it has any descendant nodes placed to the **BOD**'s right (*i.e.*, nodes with a lower priority) that are in the running status; if so, the selector will abort them all and will tick the **BOD**; otherwise, the EDBT's execution continues normally.

Therefore, the task node EXPLORE is aborted and the **BOD** is ticked, which immediately ticks its child. Observe that the task node FOLLOW uses the reference stored in *blackboard*["target"] as a parameter. In addition note that, due to (1), nothing will occur if *blackboard*["target"] is updated again while the task node FOLLOW is in the running status.

3. Coordination nodes

In this section, we will present the first part of the contribution of this work: an extension to EDBTs consisting of three new types of nodes called *coordination nodes*, and the intuition behind their use. The algorithms and implementation details will be presented in Section 6. These nodes, together with the methodology that we will present in the next section, facilitate the design and implementation of NPCs that have to coordinate with each other. For the rest of this paper, we will use the terms "NPC" and "agent" interchangeably.

3.1. Messages and requests

Coordination nodes allow agents to send and receive messages that encapsulate *requests*. A request from a sender `s` to a receiver `r` implies that `s` is requesting `r` to do something that `r` can do. In terms of behavior trees, `s` wants `r` to execute a certain subtree in `r`'s EDBT. A request is a pair $\texttt{req} = [\texttt{type}, \texttt{parameters}]$ such that `type` is a string that determines the subtree in `r`'s EDBT that `s` wants `r` to execute, and `parameters` is a potentially empty tuple $(p_0, \ldots, p_n)$ representing the parameters that customize the request. For example, a request to protect a target could be represented by $\texttt{req} = [\text{"protect""}, (\textit{target})]$.

Formally, a message from `s` to `r` is a 4-tuple $\texttt{msg} = (\texttt{s}, \texttt{req}, \texttt{c}, \texttt{t})$ where `req` is a request, `c` is a condition that `r` must satisfy to execute `req`, and `t` is a number of milliseconds after which `msg` times out and must be discarded. For example, a *commander* NPC could request the *soldier* NPC to protect a certain *target*, only if the *soldier* has more than 50% of its health points and with a timeout of 1000 milliseconds, by sending the message $(\textit{commander}, [\text{"protect""}, (\textit{target})], \textit{soldier}.\text{CURRENTHEALTH}() > \textit{soldier}.\text{MAXHEALTH}()/2, 1000)$[4]. In case the receiver doesn't need to satisfy any condition, this can be represented by the boolean value true.

All received messages are stored in the receiver's *mailbox* (a priority queue) until they are selected or discarded. When an agent is checking its mailbox, it will discard all the messages that have already timed out and—if possible—will select one that is *acceptable, i.e.*, a message with a condition it satisfies.

3.2. Request Handler nodes

Whenever an agent selects an acceptable message from its mailbox, the encapsulated request will be handled by one of the *Request Handler* (**RH**) nodes in its EDBT. This class of nodes is a specialization of the **BOD**. Recall that a **BOD** has a single composite or task node as a child and is composed of a blackboard key, a condition involving that key, and an abort rule. The only difference between both is that in the **RH** nodes the only customizable parameter is the blackboard key (a string), which represents a request type and is denoted `type`. In particular, the condition involving the key is checking if *blackboard*[`type`] has a value and its abort rule is lower-priority. This implies that, abstracting from the request protocol, an **RH** node works exactly as the **BOD** from Example 1.

An **RH** node associated with the blackboard key `type` is in charge of handling requests of that type, regardless of their parameters. Given a message $\texttt{msg} = (\texttt{s}, \texttt{req}, \texttt{c}, \texttt{t})$ with $\texttt{req} = [\texttt{type}, \texttt{parameters}]$ sent to a receiver `r`, the subtree below the **RH** node associated with `type` in the receiver's EDBT corresponds to the behavior that the sender `s` wants `r` to execute. For this reason, to be able to execute different types of requests (*e. g.*, "follow"", "protect"") an agent's EDBT must have different **RH** nodes, one for each type.

Whenever an **RH** node is ticked, if *blackboard*[`type`] has no value (*i.e.*, its condition is not met), it starts observing that blackboard key waiting for a request $\texttt{req} = [\texttt{type}, \texttt{parameters}]$ to be stored.

Behavior 1 (Handling a request)**.**
Whenever an **RH** node is notified because *blackboard*[`type`] changed:

[4] In this paper some concepts (*e. g.*, NPCs/agents, nodes) will be treated as objects of a programming language that has such data type. As is usual in object-oriented programming languages the *dot* notation will be used to access an object's properties and methods. That is, a property *p* of an object referenced by the variable *obj* will be denoted *obj.p*. In the same way, a method *m*() of the object *obj* will be denoted *obj.m*().

1. The **RH** node reevaluates its condition, *i.e.*, checks if *blackboard*[type] has a value.
   - If *blackboard*[type] has a value and there are nodes in the running status with lower priority:
     2. Those nodes are aborted.
     3. The **RH** node is ticked.
     4. The **RH** node ticks its child, *i.e.*, the root of the subtree that s wants r to execute.
   - Otherwise, if *blackboard*[type] has no value or there are no nodes in the running status with lower priority:
     2′. The execution of r's EDBT continues normally.

Given that each agent could have a different subtree below the **RH** node associated to type in its EDBT, different classes of NPCs could respond to the same request type in different ways. While an **RH**'s subtree is being executed, req = [type, parameters] will remain stored in *blackboard*[type] and, therefore, the request's parameters will be accessible by the nodes in the subtree.

In this work, we propose two classes of requests: *soft* and *hard*. Soft requests are useful when the sender wants the receivers to execute a certain subtree while the sender proceeds with its individual behavior regardless of what the receivers do. Take into account that some receivers may not be able to select the message from their mailbox before it times out if they cannot satisfy the message's condition, or if they are busy executing another request or some uninterruptible behavior. On the other hand, hard requests are useful when the sender needs to execute some behavior that depends on the receivers' commitment to actually executing a certain subtree. Hence, a hard request needs the confirmation of enough receivers before the execution of both parties' subtrees begins.

3.3. Soft Request Sender node

An agent can send messages that encapsulate a soft request through a *Soft Request Sender* (**SRS**) node, schematized in Fig. 2. This class of node is a task node (leaf) customizable with a request req, a set of receivers $r_1, \ldots, r_n$, a condition c that the receivers must satisfy to execute req, and a number of milliseconds t after which the message (and the request) time out.

Behavior 2 (Sending and receiving a soft request)**.**
Whenever an **SRS** node is ticked (see α in Fig. 2):

1. A message msg = (s, req, c, t) with req = [type, parameters] is sent to each receiver's mailbox (see β).
2. The **SRS** node returns success to its parent.
3. The execution of s's EDBT continues normally.
4. For each agent that selects msg from its mailbox (see γ):
   5. req is stored in its *blackboard*[type].
   6. Its **RH** node associated to type is notified (see δ).
   7. Behavior 1 is executed.

Observe that the **SRS** node always returns success to its parent. The reason is that the **HRS** always sends the messages successfully, and the sender can proceed with its individual behavior regardless of what the receivers do. Differently from an **HRS** node, presented below, an **SRS** node does not need the confirmation from the receivers. Another possible implementation for the **HRS** node would be to return failure when the set of receivers $r_1, \ldots, r_n$ is empty, in which case the EDBT programmer may want to handle the failure with additional nodes. Since soft requests are independent, this alternative implementation would not affect the rest of the protocol.

3.4. Hard Request Sender node

On the other hand, an agent can send messages that encapsulate a hard request through *Hard Request Sender* (**HRS**) nodes, schematized in Fig. 3. This class of node is a decorator whose only child is the root of a subtree that represents part of the sender's behavior that depends on the commitment of the receivers. **HRS** nodes are customizable with the same elements as **SRS** nodes and, also, a *quorum* q. Differently from soft requests, before the execution of both parties' subtrees begins, hard requests need the confirmation of certain receivers until the quorum specified in q is met (*e. g.*, at least a certain number of receivers or all the receivers from a list).

Behavior 3 (Sending and receiving a hard request)**.**
Whenever an **HRS** node is ticked (see α in Fig. 3):

1. A message msg = (s, req, c, t) with req = [type, parameters] is sent to each receiver's mailbox (see β).
2. The **HRS** node waits until q is met or t elapses.
3. Each agent that selects msg from its mailbox (see γ):
   4. Sends a confirmation to s (ver δ).
   5. Continues executing its EDBT normally.
4. If q is met before t elapses:

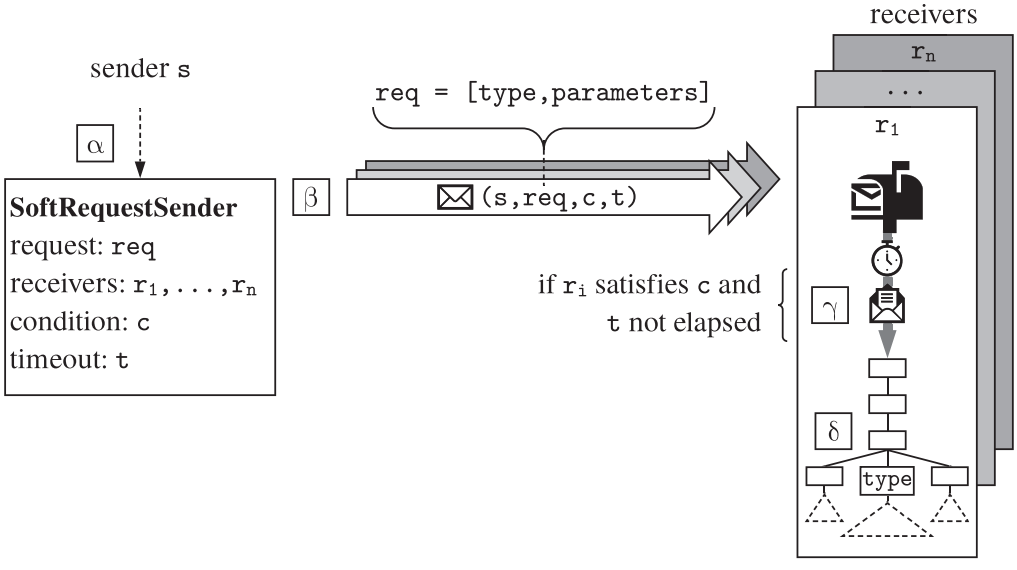

Fig. 2. Outline of a soft request from agent s to agents $r_1, \ldots, r_n$.

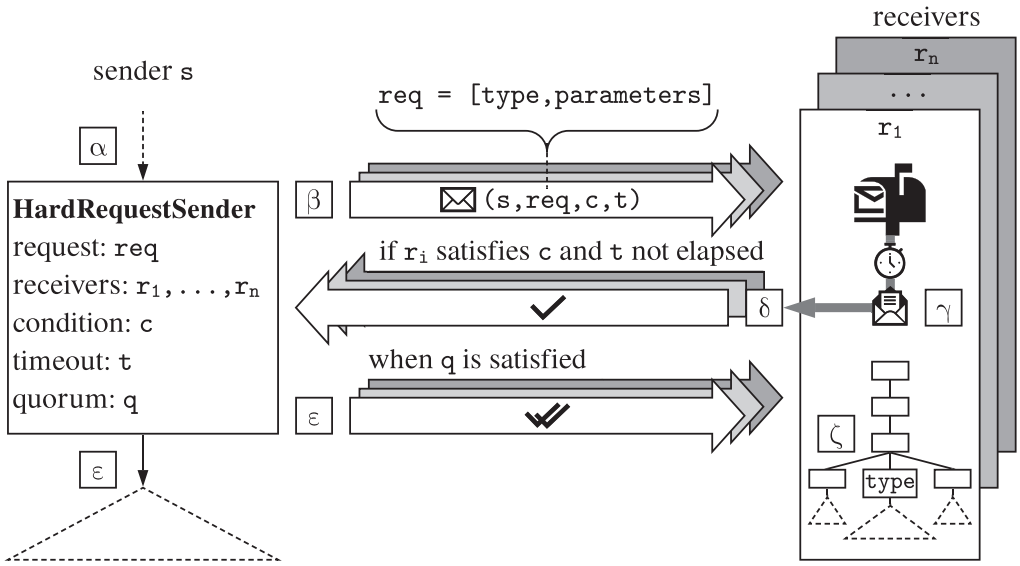

Fig. 3. Outline of a hard request from agent s to agents $r_1, \ldots, r_n$.

6. The **HRS** node ticks its child, *i.e.*, the root of s's subtree that depends on the commitment of the receivers.
7. A reconfirmation is sent to the receivers that previously confirmed (see ε).
8. For each agent that receives a reconfirmation:
   9. req is stored in its *blackboard*[type].
   10. Its **RH** node associated to type is notified (see ζ).
   11. Behavior 1 is executed.

5. Otherwise, if t elapses before q is met:
   6′. The **HRS** node returns failure to its parent.
   7′. The execution of s's continues normally.

Note that, independently of whether the request is soft or hard, it will be handled by the corresponding **RH** node in the receiver's EDBT. The class of request only determines the agents' protocol before the request is executed. Differently from soft requests, when selecting a message that encapsulates a hard request, the nodes in the running status are not immediately aborted. The execution of the receiver's EDBT continues normally and the interruption only occurs if it receives the corresponding reconfirmation.

3.5. Mailbox

Messages received by an agent are handled by the method checkMailbox. As will be explained in the next section, this method is repeatedly and concurrently called by a service node in the agent's EDBT. Whenever checkMailbox is called, it discards from the agent's mailbox all messages that have timed out and—if possible—selects one that is acceptable according to a certain criterion. For example, an NPC may prioritize messages that are closer to time out, prioritize hard requests over soft requests, prioritize certain senders over others, etc.

In our proposal, by default, agents that follow the request protocol will be committed to the coordinated behaviors they are part of. For this reason, to avoid breaking their commitment due to an interruption caused by other incoming requests, agents will automatically disable checkMailbox while:

1. executing an **RH** node's subtree,
2. waiting for an **HRS** node's quorum to be met and executing the subtree below, and
3. waiting for the reconfirmation of a hard request

In the first case, checkMailbox is automatically re-enabled after the execution of the subtree is finished. In the second case, if the request times out before the quorum is met, checkMailbox is automatically re-enabled. Otherwise, if the quorum is met in time, the method will be re-enabled after the execution of the **HRS** node's subtree is finished. In the third case, if a message with timeout t was sent at t_{msg} and the confirmation was sent at t_{conf}, the receiver will have to wait at most $t - (t_{conf} - t_{msg})$ milliseconds for the reconfirmation. If the request times out before the reconfirmation arrives in time, checkMailbox is automatically re-enabled. Otherwise, if the reconfirmation arrives, the method will be automatically re-enabled after the execution of the corresponding **RH** node's subtree is finished, as stated in the first case.

To bypass this default behavior, in the next section, we will provide tools for the EDBT designer to be able to make agents manually disable and re-enable checkMailbox when necessary. For example, the EDBT designer may want to protect from interruptions a certain critical subtree inside an agent's *individual behavior* (*i.e.*, the part of its EDBT that does not correspond to the execution of requests). In addition, the EDBT designer may want to deprotect from interruptions a subtree below an **RH** node that is considered non-critical. Considering this, a node in an agent's EDBT will be called *non-critical* if it can be ticked while checkMailbox is enabled. Therefore, by default, the nodes in the agent's individual behavior are non-critical and the nodes in an **RH** node's subtree and an **HRS** node's subtree are critical.

In this section, we presented the intuition behind coordination nodes and the request protocol, which will be formalized in Section 6 through algorithms. However, before such formalization, in the next section, we will continue with a methodology for using coordination nodes adequately.

4. Methodology

In this section, we will present a methodology for adequately structuring each agent's EDBT using the coordination nodes while following some desirable principles. In line with one of the reasons that originally motivated the use of behavior trees in the industry, this methodology will make the resulting EDBTs visually intuitive. We consider that, to achieve this high-level quality factor, each agent's EDBT must follow these principles:

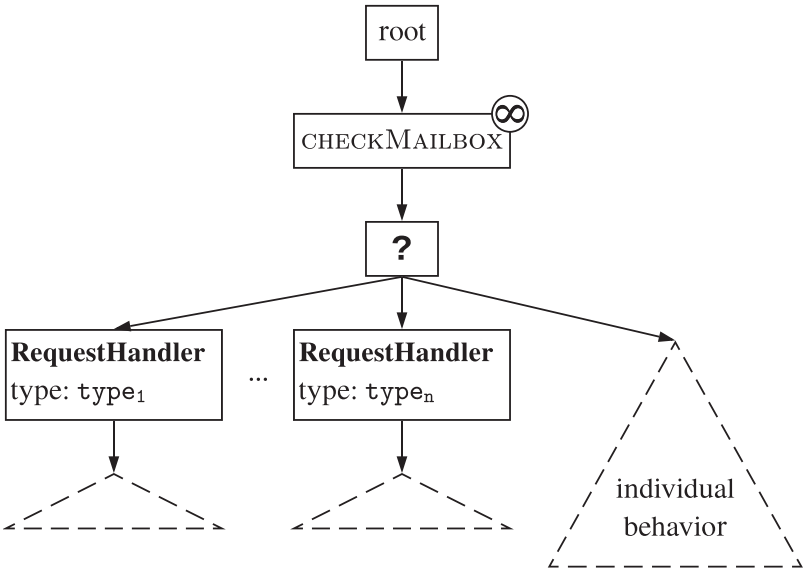

Fig. 4. Proposed EDBT structure for agents using coordination nodes.

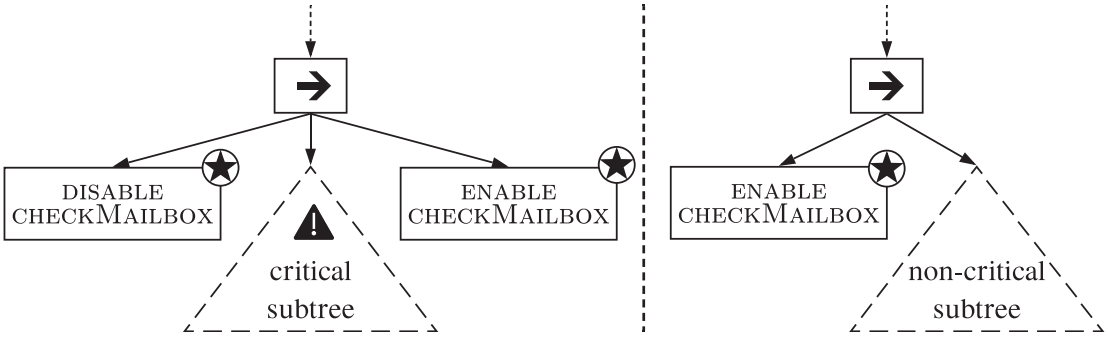

Fig. 5. The node structure on the left illustrates a sequencer (represented by a right arrow) that disables checkMailbox, executes a critical subtree, and then re-enables the method. The node structure on the right illustrates a sequencer that re-enables checkMailbox and then executes a non-critical subtree.

1. The logic of the execution of the requests is separate from the agent's individual behavior.
2. There is a visually explicit priority order between the different types of requests that the agent can handle.
3. Non-critical nodes in the running status in the agent's individual behavior can be aborted by all requests.
4. Non-critical nodes in the running status in the subtrees below **RH** nodes can only be aborted by higher-priority requests.

Regarding the last two principles, further bellow we will show how the EDBT designer can make an agent manually disable and re-enable checkMailbox to bypass its default behavior. That is, checkMailbox can be re-enabled beforehand during the execution of a subtree below an **RH** node, which causes the corresponding nodes to be non-critical; also, checkMailbox can be disabled (and then re-enabled) during the execution of the agent's individual behavior, which causes the corresponding nodes to be critical.

Fig. 4 depicts the EDBT structure that an agent that uses coordination nodes should have to follow the aforementioned principles. Consider that only relevant nodes are shown; clearly other nodes can be added as necessary.

All the **RH** nodes and the root of the agent's individual behavior should be children of a selector, which will be referred to as the *main selector*. The individual behavior should be placed to the right, and the **RH** nodes should be arranged in descending order of priority. This allows principles 1 and 2 to be satisfied. Also, given that **RH** nodes use the abort rule lower-priority, such order allows principles 3 and 4 to be satisfied. Note that, if an agent is executing a request $\texttt{req}_1$ and selects a message encapsulating another request $\texttt{req}_2$ with the same priority (*i.e.*, of the same type), $\texttt{req}_1$ will not be interrupted by $\texttt{req}_2$.

Above the main selector, there must be a service node that, after being ticked for the first time, repeatedly and concurrently calls the method checkMailbox at a frequency defined by the EDBT designer. When needed, a subtree in the agent's individual behavior that is considered critical can be protected from interruptions by using the node structure illustrated in Fig. 5 (left). Similarly, a subtree below an **RH** node or an **HRS** node that is considered non-critical can be deprotected from interruptions by using the node structure illustrated in Fig. 5 (right). As will be explained in Section 6, the disableCheckMailbox and enableCheckMailbox task nodes can be implemented by simply changing a variable's value.

Resuming with Fig. 4, any necessary nodes can be placed between the root and the main selector as long as the service node that calls checkMailbox is among them. For instance, a *loop decorator* may be necessary if in the used implementation the root node does not automatically re-tick its child whenever it returns a completion status. However, take into account that nodes that are not placed below the main selector will not be children of the **RH** node's first composite ancestor and, hence, cannot be aborted by incoming requests.

Considering the proposed structure, there are two different situations in which an **RH** node can be ticked:

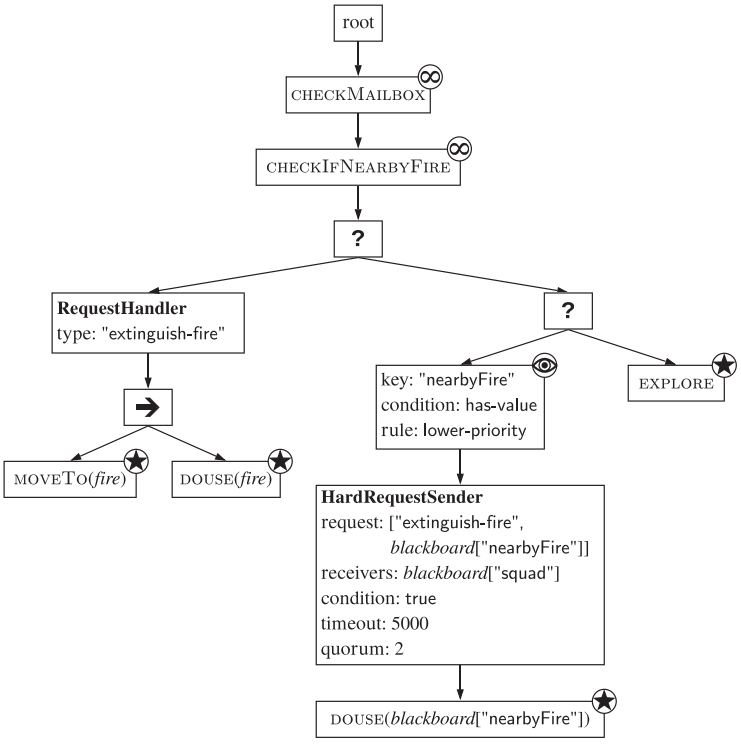

Fig. 6. EDBT for the firefighter NPCs from Example 2.

1. When the EDBT aborts its nodes in the running status to execute a request corresponding to that **RH** node (see Behavior 1), or
2. When the main selector is iterating over its children until one returns success.

In the second case, the **RH** node will always return failure since no request $\texttt{req} = [\texttt{type}, \texttt{parameters}]$ will be stored in *blackboard*[type].

Differently from **RH** nodes, **SRS** and **HRS** do not have placement restrictions besides those corresponding to tasks and decorators, respectively. They can be placed inside the agent's individual behavior or even inside the subtrees below **RH** nodes, creating the possibility of nested requests. An EDBT should not have more than one **RH** node associated with the same request type; otherwise, more than one node could simultaneously be notified by the blackboard, which would cause undesired results. On the contrary, multiple **SRS** and **HRS** nodes that send messages which encapsulate requests of the same type are allowed. If an agent selects a message encapsulating a request for which it does not have a corresponding **RH** node, the request will simply have no effect.

Differently from the execution of a receiver's EDBT, which continues normally while waiting for a reconfirmation, the execution of a sender's EDBT temporarily stops in the **HRS** while waiting until the quorum is met or the request times out. The reason is that, as we will see in Section 6, the wait for a reconfirmation occurs inside the method checkMailbox (which is executed concurrently), whereas the wait for a confirmation occurs inside the **HRS** node. A workaround to avoid this wait is to use a parallel node as the **HRS** node's parent. This allows agents to send messages that encapsulate a hard request and wait for the confirmations while executing another subtree in parallel.

5. Application example

In this section, we present an application example in which a specific coordinated behavior is modeled using coordination nodes and the methodology proposed in the previous section. This example aims to show how to use our proposal in a concrete scenario. For this reason, some elements of the application that are not part of the coordination have been simplified.

Example 2. Consider a scenario in which a squad of firefighter NPCs has the duty of extinguishing the fires caused by the player. A fire can be extinguished only if at least three firefighters simultaneously douse it for a certain amount of time. Hence, the firefighters need to coordinate to avoid focusing on different fires. The desired coordinated behavior is the following: the firefighters should explore the terrain individually; when someone finds a fire it should request two other firefighters to move to that location and help extinguish it.

Fig. 6 depicts a possible EDBT for each agent, which models the desired behavior and follows the methodology proposed in the previous section. We assume that the root automatically re-ticks its child whenever it returns a completion status; otherwise, a loop decorator can be placed below the root. Below the service node that calls checkMailbox, there is another service node that repeatedly calls the method checkIfNearbyFire, which is in charge of updating the agent's blackboard key "nearbyFire": if *blackboard*["nearbyFire"] has no value and there is a fire within a certain radius from the agent, checkIfNearbyFire stores a reference to the fire; also, checkIfNearbyFire deletes that reference from *blackboard*["nearbyFire"] if the fire is extinguished.

The main selector's right child, another selector, is the root of the agent's individual behavior. When ticked, the **BOD** below that selector checks whether *blackboard*["nearbyFire"] has a value. If that is not the case, the **BOD** starts observing the key "nearbyFire" and returns failure to the selector, which then ticks the task node explore. When this occurs, the agent explores the terrain indefinitely until the task node explore...

a. is aborted by the **BOD** (*i.e.*, the agent found a fire, whose reference was stored in *blackboard*["nearbyFire"] by checkIfNearbyFire), or

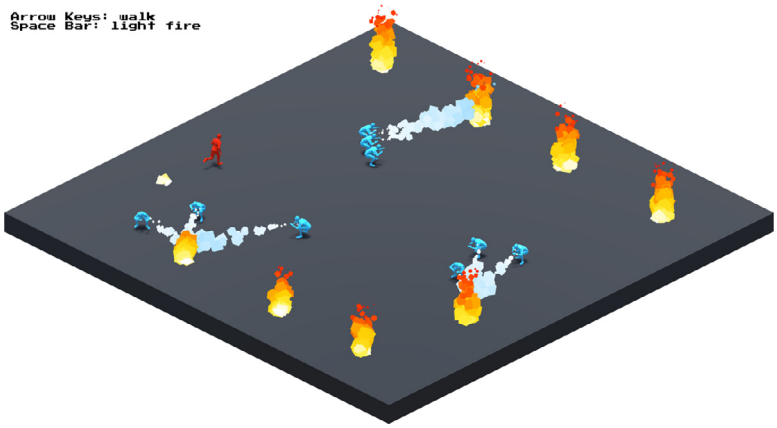

Fig. 7. A screenshot of the full implementation of Example 2. The light blue NPCs coordinate with each other, as described in the example, to extinguish the fires caused by the red character, which is controlled by the player. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

b. is aborted by an incoming request (*i.e.*, another agent found a fire).

When the first scenario occurs, the **BOD** ticks the **HRS** node. Observe that the hard request sent through this node is composed of the request type "extinguish-fire" and the parameter *blackboard*["nearbyFire"] (*i.e.*, the reference to the found fire). Also, the node contains a list of receivers stored in *blackboard*["squad"], a condition specified through the boolean value true, a timeout of 5000 milliseconds, and a quorum of 2. When ticked, the **HRS** node sends the message $\texttt{msg} = (\texttt{s}, [\text{"extinguish-fire"}, (\textit{blackboard}[\text{"nearbyFire"}])], \texttt{true}, 5000)$ to all the agents in *blackboard*["squad"], where s is the sender agent. Given that the message's condition is true, the receivers do not need to satisfy an actual condition to confirm the hard request. If the sender does not receive two confirmations before the timeout elapses, the **HRS** node returns failure to the **BOD**, the **BOD** returns failure to the selector, and the selector re-ticks the task node EXPLORE. Otherwise, if the quorum is met, a reconfirmation is sent to the agents that confirmed the request and the **HRS** node ticks its child. The task node DOUSE makes the agent start dousing the fire whose reference is stored in *blackboard*["nearbyFire"]. When the fire is finally extinguished, the task node returns success to the **HRS** node, the **HRS** node returns success to the **BOD**, and so on, until the root receives this completion status and re-ticks its child.

Whenever an agent that is exploring the terrain receives and selects a message that encapsulates a request whose type is "extinguish-fire", the **RH** node below the main selector aborts the task node EXPLORE and ticks its child. This sequencer ticks the task nodes MOVETO and DOUSE, which make the agent move to a certain distance from the fire—whose reference was sent as a parameter—and douse it until it is extinguished.

Recall that the **RH** node's subtree and the **HRS** node's subtree cannot be aborted by other incoming requests since the method CHECKMAILBOX is temporarily disabled by the request protocol.

The previous example was developed in the game development engine *Unity* using the implementation of the coordination nodes and the request protocol that will be presented in the next section. A playable version of the example[5] (see Fig. 7) and the whole Unity project[6] (containing the source code and the assets) are publicly available and can be downloaded.

6. Algorithms and implementation details

In this section, we present the main algorithms of our proposal and some relevant implementation details. An implementation of the coordination nodes and the request protocol for the game development engine *Unity* is publicly available at https://github.com/ramiroagis/CoordEDBT. The repository contains a step-by-step tutorial on how to use coordination nodes based on the application example presented in the previous section. This implementation extends *NPBehave*[7], an open-source EDBT library for Unity. Section 6.1 specifies the effects of ticking **HRS** and **SRS** nodes. Section 6.2 includes the method CHECKMAILBOX in charge of selecting acceptable messages. Section 6.3 specifies how notifications from the blackboard tick **RH** nodes. Finally, Section 6.4 presents some implementation details about this extension.

6.1. Sending a request

This section presents the main algorithms from a sender's perspective. First, Algorithm 1 specifies the effects of ticking an **SRS** node.

Algorithm 1 TICK an **SRS** node

Input: *receivers*, *agent*, *request*, *condition*, *timeout*
Output: success
1: **for each** *receiver* ∈ *receivers* **do**
2: *message* ← CREATEMESSAGE(*agent*, *request*, *condition*, *timeout*)
3: *receiver*.RECEIVEMESSAGE(*message*)
4: **end for**
5: **return** success

A method's input variables are accessible by it even though they are not specified as parameters. In particular, the *receivers*, the *request*, the *condition*, and the *timeout* are stored in the **SRS** node when it is created, whereas *agent* is the agent that is executing the EDBT (*i.e.*, the sender). Whenever an **SRS** node is ticked, a message

[5] https://github.com/ramiroagis/CoordEDBT/blob/master/Firefighters Example - Executable.zip.

[6] https://github.com/ramiroagis/CoordEDBT/blob/master/FirefightersExample-UnityProject.zip.

[7] https://github.com/meniku/NPBehave.

is created and stored in each receiver's mailbox. Then, the node returns success to its parent and the sender's EDBT's execution continues normally. Algorithm 2 specifies the effects of ticking an **HRS** node.

Algorithm 2 TICK an **HRS** node

Input: *agent*, *blackboard*, *receivers*, *request*, *condition*, *timeout*, *quorum*
Output: {success, failure}

```
1: agent.canCheckMailbox ← false
2: blackboard["confirmed"] ← CREATEEMPTYLIST()
3: for each receiver ∈ receivers do
4:     message ← CREATEMESSAGE(agent, request, condition, timeout)
5:     receiver.RECEIVEMESSAGE(message)
6: end for
7: completionStatus ← WAITFORQUORUM(request, timeout, quorum)
8: agent.canCheckMailbox ← true
9: return completionStatus
```

An agent's boolean variable *canCheckMailbox* determines at any moment if the method CHECKMAILBOX, repeatedly called by a service node in its EDBT, is enabled or disabled. Hence, to avoid breaking its commitment to the request protocol, this variable's value will be updated following the intuition explained in Section 3.5. Whenever an **HRS** node is ticked, CHECKMAILBOX is disabled and the sender's blackboard key "confirmed" is initialized with an empty list. This list will be used to store all the receivers that eventually confirm the sender's hard request. After creating and storing the messages in the receivers' mailboxes, the node starts waiting for the quorum to be met. When the wait is over, CHECKMAILBOX is re-enabled and the **HRS** node returns either success or failure to its parent depending on the output of WAITFORQUORUM, as specified in Algorithm 3.

Algorithm 3 WAITFORQUORUM(*request, timeout, quorum*)

Input: *request*, *timeout*, *quorum*, *blackboard*, *agent*, *child*
Output: {success, failure}

```
1: loop
2:     confirmedList ← blackboard["confirmed"]
3:     if ISMET(quorum, confirmedList) then
4:         for each receiver ∈ confirmedList do
5:             receiver.RECEIVERECONFIRMATION(request)
6:         end for
7:         completionStatus ← child.TICK()
8:         return completionStatus
9:     else if ISELAPSED(timeout) then
10:        return failure
11:    end if
12: end loop
```

While waiting for the quorum to be met, the **HRS** node constantly checks the list of confirmed receivers. As will be specified further below in Algorithm 6, this list is updated whenever the sender receives a confirmation from a receiver. The node waits until the quorum is met or the hard request times out. If the first case occurs, a reconfirmation is sent to each agent in the list and the **HRS** node's child is ticked; otherwise, the node returns failure to its parent. The method RECEIVERECONFIRMATION stores the request in the receiver's blackboard key corresponding to the request's type (*blackboard*[*type*] ← *request*). This will cause the receiver's blackboard to notify the corresponding **RH** node, as specified in Algorithm 8.

Algorithm 4 CHECKMAILBOX

Input: *canCheckMailbox*, *mailbox*

```
1: if canCheckMailbox and not mailbox.ISEMPTY() then
2:     message ← mailbox.SELECTMESSAGE()
3:     if message ≠ null then
4:         request ← message.GETREQUEST()
5:         if request is SoftRequest then
6:             HANDLESOFTREQUEST(request)
7:         else if request is HardRequest then
8:             sender ← message.GETSENDER()
9:             timeout ← message.GETTIMEOUT()
10:            HANDLEHARDREQUEST(sender, request, timeout)
11:        end if
12:    end if
13: end if
```

Algorithm 5 HANDLESOFTREQUEST(*request*)

Input: *request*, *blackboard*

```
1: type ← request.GETTYPE()
2: blackboard[type] ← request
```

Algorithm 6 HANDLEHARDREQUEST(*sender, request, timeout*)

Input: *sender*, *request*, *timeout*, *agent*, *canCheckMailbox*

```
1: sender.RECEIVECONFIRMATION(agent)
2: canCheckMailbox ← false
3: type ← request.GETTYPE()
4: WAITFORRECONFIRMATION(request, type, timeout)
```

Algorithm 7 WAITFORRECONFIRMATION(*request, type, timeout*)

Input: *request*, *type*, *timeout*, *blackboard*, *canCheckMailbox*

```
1: loop
2:     if ISELAPSED(timeout) then
3:         canCheckMailbox ← true
4:         break loop
5:     else if blackboard[type] = request then
6:         break loop
7:     end if
8: end loop
```

Algorithm 8 BENOTIFIED

Input: *blackboard*, *type*

```
1: if blackboard[type] ≠ null then
2:     selector ← self.GETPARENT()
3:     ticked ← selector.ABORTLOWERPRIORITY(self)
4:     if not ticked then
5:         blackboard[type] ← null
6:     end if
7: end if
```

6.2. Checking the mailbox

This section presents the algorithms regarding the mailbox checking from a receiver's perspective. Next, Algorithm 4 specifies the method CHECKMAILBOX, which is repeatedly and concurrently called by a service node in the agent's EDBT.

The method SELECTMESSAGE removes from the agent's mailbox all messages that have timed out and—if possible—selects one that is acceptable (*i.e.*, one whose condition is satisfied by the agent) according to some criterion defined by the EDBT designer. The message's request will be handled differently depending on whether it is soft or hard, as specified in Algorithms 5 and 6, respectively.

Since soft requests do not require confirmations, they are immediately stored in the receiver's blackboard key corresponding to the request's type. This makes the receiver's blackboard notify the corresponding **RH** node, as specified in Algorithm 8.

On the contrary, since hard requests do require confirmations, receiveConfirmation stores the receiver (*i.e.*, the *agent* that is executing the method) in the sender's blackboard key corresponding to the list of confirmed receivers (*blackboard*[confirmed] ← *agent*). Then, the receiver temporarily disables the method checkMailbox to avoid breaking its commitment to a confirmed request which has not yet started. Algorithm 7 specifies how the receiver waits for the reconfirmation.

Recall that, since checkMailbox is called concurrently by a service node, the confirming agent's EDBT's execution can continue normally while it awaits the reconfirmation. This wait will stop when the request times out or when the request is stored in the receiver's blackboard key corresponding to the request's type (see receiveReconfirmation in Algorithm 3). In the first case, the method checkMailbox is re-enabled.

6.3. Being notified by the blackboard

This section specifies, from a receiver's perspective, how a notification from the blackboard occurs and how it causes the corresponding **RH** node to be ticked.

When an **RH** node associated with the request type *type* is created, it is registered in the EDBT's *blackboard* as an observer for the key *type*. Hence, the **RH** node will be notified whenever *blackboard*[*type*] is modified. This will occur in two different situations: whenever a sender's *request* whose type is also *type* is stored in the receiver's blackboard (recall the sentence *blackboard*[*type*] ← *request* from Algorithms 3 and 5); and when the request's execution is finished and needs to be removed from the blackboard to prepare the **RH** node for future requests of the same type (*i.e.*, *blackboard*[*type*] ← null). Algorithm 8 specifies the effects of an **RH** node being notified by the blackboard.

Whenever an **RH** node is notified, first it verifies that it is due to a request ready to be executed. Then, the node's parent[8] will abort all its descendant nodes in the running status with a lower priority. If the methodology presented in Section 4 is used, the node's parent will be the main selector, and it will be able to abort any non-critical node in the running status inside another **RH** node's subtree to the right, or any non-critical node in the running status inside the agent's individual behavior. If at least one node is aborted, the **RH** node is ticked and abortLowerPriority returns true; otherwise, it returns false and the EDBT's execution continues normally. Note that, if no nodes are aborted, this implies that a higher-priority request was being executed.

If the **RH** node is ticked, the request will remain stored in the blackboard until the execution of the subtree below the node is finished. This allows the nodes in the subtree to fetch all the necessary parameters from the blackboard. Otherwise, the request is deleted from the corresponding blackboard key to prepare the node for future requests of the same type. Note that *blackboard*[*type*] ← null causes beNotified to be called again, but nothing happens due to its initial check. Finally, Algorithm 9 specifies the effects of ticking an **RH** node.

If *blackboard*[*type*] ≠ null, the node was ticked by a blackboard notification and, therefore, a request is ready to be executed (see Algorithm 8). In this case, the **RH** node's child is ticked, which will eventually return success or failure when the execution of the subtree is finished. Note that the method checkMailbox is disabled while the **RH** node's subtree is being executed to avoid interruptions caused by other incoming requests. Afterward, the request is deleted from the corresponding blackboard key to prepare the node for future requests of the same type. On the contrary, if *blackboard*[*type*] = null, the node was ticked by the main selector when iterating over its children. In this case, the **RH** node returns failure to its parent since no request needs to be executed.

Algorithm 9 tick an **RH** node

Input: *blackboard*, *type*, *agent*, *child*
Output: {success, failure}
1: **if** *blackboard*[*type*] ≠ null **then**
2: *agent.canCheckMailbox* ← false
3: *completionStatus* ← *child*.tick()
4: *agent.canCheckMailbox* ← true
5: *blackboard*[*type*] ← null
6: **return** *completionStatus*
7: **else**
8: **return** failure
9: **end if**

Next follows a final remark on Algorithms 2 and 3, and the method receiveConfirmation from Algorithm 6. Since multiple **HRS** nodes could be placed as descendants of a parallel node, agents could send different hard requests simultaneously. For this reason, each corresponding list of confirmed receivers should actually be stored in the blackboard key composed of the concatenation of the string "confirmed" and a unique identifier associated with each **HRS** node (*i.e.*, *blackboard*["confirmed" + *nodeId*]). Otherwise, the receivers that confirm the different hard requests would end up stored in the same list and the coordination among the agents would not occur as expected.

6.4. Implementation details and time complexity

This section presents some relevant details regarding the implementation and discusses the time complexity of the algorithms presented in the previous section.

The extension that we have presented in this paper can be used in the game development engine *Unity* by importing the C# classes that are available in the previously mentioned repository and the ones provided by *NPBehave*, the base EDBT implementation.

Among the classes that implement this extension, the ones that are of particular interest for the user are *HardRequestSender, SoftRequestSender, RequestHandler*, and *Agent*. As their names imply, the first three classes implement the coordination nodes and can be used as described by the step-by-step tutorial in the repository. To be able to use these nodes, an NPC must that inherit from the class *Agent*. By doing so, the NPC automatically follows the request protocol and has access to a mailbox and the methods disableCheckMailbox and enableCheckMailbox, which can be invoked from task nodes.

Next, we will discuss the time complexity of the algorithms in the worst-case scenario: Algorithm 1 (tick an **SRS** node) runs in linear time with respect to the number of receivers. The runtimes of Algorithm 2 (tick an **HRS** node), Algorithm 3 (waitForQuorum) and Algorithm 9 (tick an **RH** node) are bounded by that of *child*.tick, like many other nodes in any Behavior Trees implementation (*e. g.*, the *conditional decorator, blackboard observer decorator*). Algorithm 5 (handleSoftRequest) runs in constant time. Algorithm 4 (checkMailbox), Algorithm 6 (handleHardRequest), and Algorithm 7 (waitForReconfirmation) run in a fixed amount of time determined by the *timeout* variable, like other nodes in existing Behavior Trees implementations (*e. g.*, the *wait* node, the *time limit* node). Algorithm 8 (beNotified) runs in linear time with respect to the number of nodes in the EDBT. Considering this, we can con-

[8] self refers to the **RH** node that is executing the method.

clude that the coordination nodes that we propose in this paper do not increase the time complexity of the EDBTs.

7. Related work and discussion

In Section 1, we introduced a general overview of the related work on behavior trees. In this section, we will discuss the differences and similarities with two approaches that are closely related to our proposal. Also, we will discuss the benefits of using coordination nodes in comparison to hard-coding coordinated behaviors without it.

Soft and hard requests are related to FIPA's *Request-When Interaction Protocol* (FIPA: Foundation for Intelligent Physical Agents, 2002). Similar to our approach, this interaction protocol allows an *initiator* agent to request a *participant* agent to perform some action when the given precondition becomes true. However, there are some differences. First, if the participant does not *understand* the request it will initially refuse; otherwise, it will agree and wait until the precondition occurs. Although in our approach this concept of "understanding the request" is inexistent, a beforehand agreement (confirmation) is sent by the receiver of a hard request when the message is selected from its mailbox (hence, the condition is already true). Another difference is that in FIPA's interaction protocol there is no quorum or reconfirmation, and once the action is completed the participant informs the initiator about the result of the action.

In (Marzinotto et al., 2014), the authors present a unified framework for classical BTs for robotics and control applications is presented. They mention the need for extending their framework with a *Decorator˜* node that allows two or more agents to undertake a common task jointly by synchronizing parts of their BTs. This type of node should allow its subtree to be synchronized with the subtrees below the *Decorator˜* nodes corresponding to the same cooperative task that is in other agents' BTs. Whenever a *Decorator˜* node is ticked, it should communicate to the other corresponding *Decorator˜* nodes that it is ready to engage as soon as there are enough available agents to simultaneously execute the corresponding subtree. If the *Decorator˜* node is ticked when there are enough available agents, all the corresponding subtrees are simultaneously executed; otherwise, the node returns without ticking its subtree. Although the authors provide a high-level description of the desired behavior for this node, they do not formalize these intuitions or provide a concrete implementation of such behavior. Putting aside the fact that our approach is not based on classical BTs, the **HRS** decorator together with the **RH** observer decorator node and the mailbox CHECKMAILBOX serves as the extension suggested by (Marzinotto et al., 2014). The difference is that, thanks to the event-drivenness of EDBTs, when an **HRS** node sends a hard request to a set of receivers they do not necessarily have to be free to be available to carry out the coordination of subtrees. In case the cooperative task needs to be the same for all agents, as described by the authors, the subtrees below the **HRS** and **RH** nodes should simply be the same.

Another form of coordination in multi-agent systems is the coalition and community formation using cooperative game theories. For instance, in (Wahab et al., 2016), the authors investigate the problem of community-based cooperation among intelligent Web service agents by modeling the community formation problem as a Stackelberg game model. In (Asl et al., 2014), the authors propose a game-theoretic-based decision mechanism that agents can use to choose competition or cooperation strategies that maximize their payoffs. This decision mechanism could be integrated into our approach to implement a message selection criterion that maximizes the payoff (see SELECTMESSAGE from Algorithm 4). In (Hu & Leung, 2017), the authors show that agents can achieve coordination via establishing diverse stable local conventions, which indicates a practical way to solve coordination problems.

In Section 2, we mentioned that blackboards are used to store the data that needs to be referenced by the nodes in an EDBT. Since blackboards can also be shared among multiples agents and used as a synchronization mechanism, one may wonder: Are shared blackboards enough for implementing coordinated behaviors? What are the implications of that alternative? Next, we will show how the scenario from Example 2 can be implemented without coordination nodes by using regular EDBTs and a shared blackboard, and we will analyze the disadvantages of that approach.

Example 3. Each firefighter NPC has a private blackboard (*blackboard*) used to store its personal data. Also, each agent has access to another blackboard (*sharedbb*) that is shared among all the firefighters. In particular, *sharedbb*["foundFires"] is used to store a shared list that contains the references to the fires that were found by the agents. A service node in each agent's EDBT repeatedly and concurrently calls the method CHECKIFNEARBYFIRE that checks if there is a fire within a certain radius and stores a reference to it at the end of *sharedbb*["foundFires"] (only if it is not already stored). Then, the reference to the fire is used as a key in the shared blackboard, whose value is initialized to 0 to keep track of how many agents are dousing the fire (*i.e.*, *sharedbb*[*fire*] ← 0). Also, another service node in each agent's EDBT repeatedly and concurrently calls the method coordinate, specified next:

If the agent is not busy and there are available fires, the method COORDINATE stores the first reference in the list (*i.e.*, the last found fire) in *blackboard*["target"]. Like in Example 2, a **BOD** that observes the blackboard key "target" can be used to abort the task node EXPLORE and then tick the task nodes MOVETO and DOUSE, which make the agent move to a certain distance from the fire—whose reference is *blackboard*["target"]—and douse it until it is extinguished. Therefore, the sentence *blackboard*["target"] ← *target* would cause the **BOD** to be notified by the blackboard and the task node EXPLORE to be aborted. The variable *target* is then used as a key for the shared blackboard to update the number of agents that are dousing the fire. If that number becomes 3, the blackboard key is emptied and the reference to the fire is removed from the list. Finally, two additional task nodes would be required in each agent's EDBT to change the value of the variable "*busy*" accordingly, like the task nodes ENABLECHECKMAILBOX and DISABLECHECKMAILBOX from Fig. 5. Take into account that Algorithm 10 is missing the necessary checks to avoid race conditions.

Algorithm 10 COORDINATE

```
Input: busy, blackboard, sharedbb
1: if not busy and not sharedbb["foundFires"].ISEMPTY() then
2:     target ← sharedbb["foundFires"].GETFIRST()
3:     blackboard["target"] ← target
4:     sharedbb[target] = sharedbb[target] + 1
5:     if sharedbb[target] = 3 then
6:         sharedbb[target] ← null
7:         sharedbb["foundFires"].REMOVE(target)
8:     end if
9: end if
```

The coordinated behavior from the previous example is carried out by hard-coding and (unintentionally) hiding the coordination itself inside a method that is called by a service node. The problem with this ad hoc solution is that it goes against the development paradigm of behavior trees (*i.e.*, programming only the NPCs' actions and designing a tree structure that determines its decision making) and partially drives away some of the benefits of the paradigm: being visually intuitive, scalable and reusable. With this

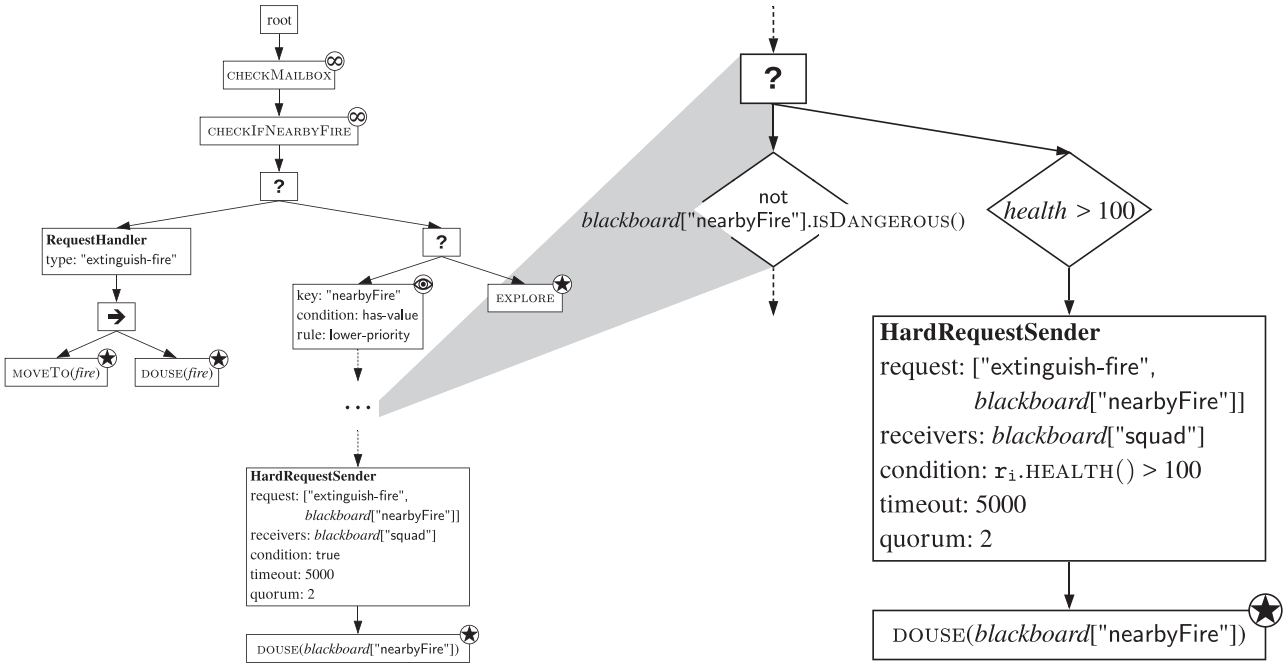

Fig. 8. The EDBT used to implement the coordinated behavior from Example 2 and adapted to consider the changes from Example 4 by adding the node structure at the right. The rhombuses represent conditional decorators.

kind of solution, the more complex the coordinated behavior, the higher the amount of hard-coding that is necessary, which essentially ruins the original purpose of behavior trees. On the contrary, by using coordination nodes—like in Example 2—not only the NPCs' domain-specific behavior is visually specified in the EDBT, but also the development paradigm of behavior trees is maintained. Even though in our proposal part of the agents' request protocol is modeled by the method CHECKMAILBOX, it is standard for every NPC in any scenario and does not directly affect the domain-specific agents' behavior.

Before concluding this section, we will modify the scenario described in Example 2 and show how the solution that uses an EDBT with coordination nodes (see Fig. 6) can be easily adapted to consider the new changes, while the ad hoc solution from the previous example requires the hard-coded method that handles the coordination to be completely reworked.

Example 4. Consider the scenario described in Example 2 and suppose that the player can now cause fires that are dangerous for the firefighters. Firefighters that stay nearby a dangerous fire can get hurt by the heat and, therefore, must have more than 100 health points before starting to douse it to avoid fainting in the process.

Fig. 8 shows how the EDBT from Fig. 6 can be adapted to consider these changes by adding five nodes between the **BOD** and the **HRS** node. The selector together with the conditional decorator below it are used to branch the EDBT into two different courses of action. When this conditional decorator is ticked, if the fire whose reference is stored in *blackboard*["nearbyFire"] is not dangerous, the **HRS** node (the same as the one in Fig. 6) is ticked and sends to the squad a message that encapsulates a hard request to extinguish the fire without condition (represented by the boolean value true). Otherwise, if the fire is dangerous, the conditional decorator returns failure to the selector, which then ticks another conditional decorator. When this conditional decorator is ticked, if the agent has more than 100 health points, the new **HRS** node is ticked and sends to the squad a message that encapsulates a hard request to extinguish the fire, which can be confirmed only by receivers with more than 100 health points. In both cases, if the quorum is met before the timeout elapses, the task node DOUSE is ticked. Note that changing the behavior corresponding to the execution of the requests is not necessary.

On the contrary, changing the ad hoc solution from Example 10 to achieve the desired coordinated behavior is not straightforward. First of all, the method CHECKIFNEARBYFIRE needs to be modified so that any fire that is found is stored either in *sharedbb*["foundDangerousFires"] or in *sharedbb*["foundHarmlessFires"] depending on whether it is dangerous. One may think that the method COORDINATE (see Algorithm 10) can be simply adapted by changing lines 1 and 2 to assign a reference to a fire to the variable "*target*" considering both the agent's health points and that either *sharedbb*["foundDangerousFires"] or *sharedbb*["foundHarmlessFires"] (or both) could be empty. However, even if the variable "*target*" has value, *blackboard*["target"] $\leftarrow$ *target* cannot be executed right away: if that fire is dangerous and at that moment there are not two more agents with more than 100 health points, that assignment would cause the agent to move to the fire and douse it vainly, potentially losing health points due to the heat. Hence, if the agent has a target, it should simply increase the value of *sharedbb*[*target*] by 1. Once the agent has a target, every time that the method COORDINATE is executed it should check if there are enough agents ready to extinguish that fire (*i.e.*, *sharedbb*[*target*] $= 3$) and, in that case, execute *blackboard*["target"] $\leftarrow$ *target*. When that occurs, the target should be removed either from *sharedbb*["foundDangerousFires"] or from *sharedbb*["foundHarmlessFires"], accordingly.

The problem with the solution described above is that if the squad consists of only three or four firefighters, two agents may end up waiting indefinitely to extinguish a dangerous fire while the rest are waiting to extinguish a harmless one. For this reason, even though firefighters should naturally focus (if possible) on dangerous fires, the method COORDINATE needs additional checks to avoid this kind of "deadlock".

In contrast, this undesired situation cannot occur with the solution that uses coordination nodes. Recall Fig. 7 and suppose the worst-case scenario in which the squad has only three firefighters and the only one with more than 100 health points initially

finds a dangerous fire. A message that encapsulates a hard request to extinguish it is sent to the other two firefighters, but they cannot satisfy the condition before the request times out and thus the quorum is not met. The **HRS** node (the one at the right) returns failure to the conditional decorator (*health* > 100), the conditional decorator returns failure to selector, the selector returns failure to the **BOD**, the **BOD** returns failure to the selector, and the selector finally ticks the task node EXPLORE. The method CHECKIFNEARBYFIRE does not cause another interruption even though that very same fire is still within range given that *blackboard*["nearbyFire"] is unaffected. Therefore, the agent does not insist with another hard request to extinguish the same fire and continues exploring in search of new ones.

The previous example showed the problems that arise when it is necessary to change a coordinated behavior that is already implemented by using an EDBT and a hard-coded method that handles the coordination. Differently from using coordination nodes, that kind of ad hoc solutions not only go against the development paradigm of behavior trees, but also show to be neither visually intuitive, nor reusable, nor scalable.

8. Conclusions and future work

In this paper, we have proposed and implemented a novel approach for agent coordination in multi-agent systems consisting of an extension to behavior trees (BTs), a behavior creation method that is popular in the video game industry. The importance of our approach is that it provides a concrete extension that facilitates the design and implementation of agents that need to coordinate with each other. For that purpose, in this work we extended event-driven behavior trees with three new types of nodes, called coordination nodes, which facilitate the design and implementation of NPCs that can coordinate with each other through a request protocol. A request implies that the sender agent wants the receiver agent to execute a certain subtree in the receiver's EDBT. In particular, **SRS** and **HRS** nodes can be used to send messages that encapsulate a soft request or a hard request to multiple agents, respectively. Independently of whether the request is soft or hard, it will be handled by the corresponding **RH** node in the receiver's EDBT. The class of request only determines the agents' protocol before the request is executed. Soft requests are useful when the sender wants the receivers to execute a certain subtree while the sender proceeds with its individual behavior regardless of what the receivers do. On the other hand, hard requests are useful when the sender needs to execute some behavior that depends on the receivers' commitment to actually executing a certain subtree. Hence, a hard request needs the confirmation of enough receivers before the execution of both parties' subtrees begins.

These nodes do not provide more "expressive power" in terms of the coordinated behaviors that could be created by using regular EDBTs. However, unlike the usual ad hoc solution of hard-coding the coordinated behavior inside a method that is repeatedly called by the EDBT, creating coordinated behaviors by using coordination nodes follows the development paradigm of behavior trees. Also, we have concluded that coordination nodes do not increase the time complexity of EDBTs.

We presented a methodology for adequately using the coordination nodes while following some desirable principles that make the resulting EDBTs visually intuitive. Also, we provided a full implementation of the coordination nodes and the request protocol for the game development engine *Unity*. This implementation was used to develop an application example for the proposed extension. Finally, we discussed the benefits of using this extension in comparison to using regular EDBTs and hard-coding the coordinated behavior inside a method that uses a shared blackboard.

Regarding future work, we plan to extend **HRS** nodes to allow NPCs to send simultaneously different types of hard requests to different sets of receivers. With this extension, the sender would have multiple quorums (one for each type) which must all be met before the coordination of the subtrees begins. Although this can also be achieved by using multiple **HRS** nodes together with some sequencer and parallel nodes we believe that, by extending **HRS** nodes as proposed, the easy of use and the intuitiveness of our proposal for those particular cases will be improved.

Also, we plan to extend our proposal with a new coordination node that allows *hard request chaining*. That is, allow an agent that selects a hard request $\texttt{req}_1$ from its mailbox to immediately send another hard request $\texttt{req}_2$ whose quorum needs to be met before confirming $\texttt{req}_1$. If $\texttt{req}_2$ receives enough confirmations for its quorum to be met, a confirmation is sent to $\texttt{req}_1$'s sender; then, if the reconfirmation for $\texttt{req}_1$ is eventually received, it is retransmitted to $\texttt{req}_2$'s receivers.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work has been partially supported by CONICET, Universidad Nacional del Sur, Argentina, and by PGI-UNS (grants 24/ZN32, 24/N046).

References

2K Games (2007). Bioshock.
Asl, E. K., Bentahar, J., Mizouni, R., Khosravifar, B., & Otrok, H. (2014). To compete or cooperate? This is the question in communities of autonomous services. *Expert Systems with Applications, 41*(10), 4878–4890.
Champandard, A. J., & Dunstan, P. (2012). The behavior tree starter kit. *Game AI Pro: Collected Wisdom of Game AI Professionals*, 72–92.
Colledanchise, M., & Ögren, P. (2014). How behavior trees modularize robustness and safety in hybrid systems. In *Intelligent robots and systems (iros 2014), 2014 ieee/rsj international conference on* (pp. 1482–1488). IEEE.
Colledanchise, M., Parasuraman, R. N., & Ogren, P. (2018). Learning of behavior trees for autonomous agents. *IEEE Transactions on Games*.
Electronic Arts (2008). Spore.
FIPA: Foundation for Intelligent Physical Agents (2002). FIPA request when interaction protocol specification. http://www.fipa.org/specs/fipa00028/SC00028H.pdf.
Flórez-Puga, G., Gomez-Martin, M. A., Gomez-Martin, P. P., Díaz-Agudo, B., & González-Calero, P. A. (2009). Query-enabled behavior trees. *IEEE Transactions on Computational Intelligence and AI in Games, 1*(4), 298–308.
Hu, D., Gong, Y., Hannaford, B., & Seibel, E. J. (2015). Semi-autonomous simulated brain tumor ablation with Ravenii surgical robot using behavior tree. In *Robotics and automation (icra), 2015 ieee international conference on* (pp. 3868–3875). IEEE.
Hu, S., & Leung, H.-f. (2017). Achieving coordination in multi-agent systems by stable local conventions under community networks.. In *Ijcai* (pp. 4731–4737).
Isla, D. (2005). Managing complexity in the Halo2 AI. *Game developer's conference, 2005*.
Isla, D. (2008). Building a better battle. *Game developers conference, san francisco*: 32.
Johansson, A., & Dell'Acqua, P. (2012). Emotional behavior trees. In *Computational intelligence and games (cig), 2012 ieee conference on* (pp. 355–362). IEEE.
Lemaitre, J., Lourdeaux, D., & Chopinaud, C. (2015). Towards a resource-based model of strategy to help designing opponent AI in RTS games. In *7th international conference on agents and artificial intelligence (icaart 2015): 1* (pp. 210–215).
Lim, C.-U., Baumgarten, R., & Colton, S. (2010). Evolving behaviour trees for the commercial game DEFCON. In *European conference on the applications of evolutionary computation* (pp. 100–110). Springer.
Marzinotto, A., Colledanchise, M., Smith, C., & Ögren, P. (2014). Towards a unified behavior trees framework for robot control. In *Robotics and automation (icra), 2014 ieee international conference on* (pp. 5420–5427). IEEE.
Microsoft Game Studios (2004). Halo 2.
Microsoft Game Studios (2007). Halo 3.
Ogren, P. (2012). Increasing modularity of UAV control systems using computer game behavior trees. In *Aiaa guidance, navigation, and control conference* (p. 4458).
Orkin, J. (2006). Three states and a plan: the AI of FEAR. In *Game developers conference: 2006* (p. 4).

Proskurnikov, A. V., & Cao, M. (2017). Differential inequalities in multi-agent coordination and opinion dynamics modeling. *Automatica, 85*, 202–210.
Sakurama, K., & Ahn, H.-S. (2020). Multi-agent coordination over local indexes via clique-based distributed assignment. *Automatica, 112*, 108670.
Shoulson, A., Garcia, F. M., Jones, M., Mead, R., & Badler, N. I. (2011). Parameterizing behavior trees. In *International conference on motion in games* (pp. 144–155). Springer.
Square Enix (2018). Just Cause 4.
Ubisoft (2016a). Far Cry Primal.
Ubisoft (2016b). Tom Clancy's The Division.
Wahab, O. A., Bentahar, J., Otrok, H., & Mourad, A. (2016). A stackelberg game for distributed formation of business-driven services communities. *Expert systems with applications, 45*, 359–372.
Wang, X., Zeng, Z., & Cong, Y. (2016). Multi-agent distributed coordination control: Developments and directions via graph viewpoint. *Neurocomputing, 199*, 204–218.
Warner Bros. Interactive Entertainment (2005). F.E.A.R.
Yannakakis, G. N. (2012). Game AI revisited. In *Proceedings of the 9th conference on computing frontiers* (pp. 285–292). ACM.
Yannakakis, G. N., & Togelius, J. (2018). *Artificial intelligence and games*. Springer.
Zhang, H., & Su, S. (2019). A hybrid multi-agent coordination optimization algorithm. *Swarm and Evolutionary Computation, 51*, 100603.
Zhao, W., Li, R., & Zhang, H. (2017). Leader–follower optimal coordination tracking control for multi-agent systems with unknown internal states. *Neurocomputing, 249*, 171–181.
Zhao, W., & Zhang, H. (2019). Distributed optimal coordination control for nonlinear multi-agent systems using event-triggered adaptive dynamic programming method. *ISA Transactions, 91*, 184–195.
Zou, Y., Su, X., Li, S., Niu, Y., & Li, D. (2019). Event-triggered distributed predictive control for asynchronous coordination of multi-agent systems. *Automatica, 99*, 92–98.