Policy-Driven Behavior Trees

Simulating Autonomous Adaptive Robotic Behavior
in a Virtual Test Environment Using Policy-Driven Behavior Trees

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science (MSc), 45 ECTS

Utrecht, March 24, 2018

Written by

Julius van der Kwast

Student number 3386821

Supervised by

Prof. Dr. Mr. Henry Prakken *(Utrecht University)*

Prof. Dr. John-Jules Meyer *(Utrecht University)*

Abstract

The research discussed in this paper aims to simplify the creation and modification of adaptive behavior for a robot. The following research question was formulated. *How can a framework for robots be created that simplifies the off-line creation and modification of adaptive behavior?* To answer this question, two sub-questions answer how this can be technically realized and how this framework can be operationally embedded. The technical realization has been done by writing policies in Drools, the creation of behavior trees in a virtual environment created in Unity and their combination in a policy engine. TNO developed the *Policy Engine TNO (PET)* that allowed the policies and the behavior trees to communicate with each other, creating a policy-driven behavior tree. Both policies and behavior trees are intuitive to use and encourage non-experts to experiment with the creation and alteration of adaptive behavior strategies. The framework entails this combination and was specifically developed for a house search mission, simulated in a 3D virtual test environment. An ontology was written in the policy engine so that the robot is able to classify objects in the virtual environment in relation to other objects. The robot may recognize a *weapon* in the virtual environment and know that this is classified as a *dangerous object*. With this ontology, several policies were created to function as a conflict-solving mechanism and allow the robot to follow these policies and overrule any previously issued policies. The scenario that followed from this house search mission was carefully created in collaboration with domain experts that are closely working together with TNO. With the creation of this scenario and a simulation run to prove that the integration of policies and a behavior tree works, the sub-question regarding operationally embedding was fulfilled.

Contents

List of Figures

1 Introduction

Robots play an ever-increasing role in human society. A robot is defined as a machine, programmable by a human, capable of carrying out difficult and complex sets of actions automatically. We are slowly shifting towards a future where robots are (partially) replacing the actions performed by humans. Traditional vacuum cleaners are replaced by automated, self-driving robot vacuum cleaners, which use different sensors to find and clean dirt without the need for human interaction (Abramson, Levin, & Zaslavsky, 2006). The continuous development of autonomous cars will gradually shift towards a future which requires less and less human involvement, where the autonomous car can sense its environment and navigate without the need for human input. Industrial robots are automated, programmable robot systems that are used for manufacturing. Typical applications of an industrial robot include painting, picking up objects and dropping them at the desired place, product inspection, assembly and testing (Nof, 1999).

In basic robotics, machines are designed to do the tasks specified to them. These are often static, pre-defined tasks and they have no way of dealing with unexpected situations. Basic robotics may be, for example, the manufacturing process of a watch by fitting small pieces of the watch together. Advanced robots are designed to be adaptive and autonomous. Adaptive behavior means the robot can respond according to the changing environment and autonomous means that the robot is capable of making decisions on its own (Martius, Der, & Ay, 2013). The autonomous car can recognize, for example, a kid crossing the street, adjusting its speed to avoid collision.

This shifting towards a robot-centric future is gradually taking place and is limited by the possibilities and limitations of the robot itself. While robots may largely or completely replace humans in some areas, there are still many areas where the role of humans is central and robots fulfill a supplementary role. The activities performed by the fire brigade could greatly benefit from the presence of robots. Consider a scenario where a house burns down and people are trapped inside. Quick decision-making skills are mandatory and could save lives. Current technology is not ready yet to replace firemen, who are required to enter the building, making decisions based on an ever-changing dynamic environment. Making a mistake could be dangerous and life-threatening. Robots, however, could greatly increase the success chance of a mission. A drone could quickly

elevate towards the upper floors of a building, spotting victims trapped inside the building by using cameras. An unmanned ground vehicle (UGV for short) is a vehicle that operates while in contact with the ground and has no onboard human presence. An unmanned ground vehicle could accompany firemen inside the building, moving through places that are too small or too hot for firemen to enter, exploring new areas and communicating its findings.

These are scenarios where the collaboration between humans and robots is crucial, greatly influencing the outcome of a mission. Situations like these require fast and accurate decision-making skills, where a mistake could have great consequences. It is exactly here that it is very important that the robot can adapt its behavior based on developments during the mission, resulting in new, more effective behaviors. Human-Robot Interaction (HRI) and Human-Robot Communication (HRC) in particular are of primary importance when dealing with human-robot teams that operate outside of production lines (Klingspor, Demiris, & Kaiser, 1997). The common thread between robots and humans is that they form a team with common goals.

1.1 Human-Robot Teaming

When working in high-intensity domains, tasks are often well-defined but involve complex coordination in stressful and dynamic situations. An example of such an environment is where human-robot teams assist in disaster response efforts, as is the case in the TRADR-project (Long-Term Human-Robot Teaming for Robot-Assisted Disaster Response). Using a proven-in-practice user-centric design methodology, TRADR develops technology for human-robot teams to assist in disaster response efforts, with the challenge to make experience persistent over time. In the TRADR scenario, various kinds of robots collaborate with human team members to explore the environment and gather physical samples. Throughout this collaborative effort, the team gradually develops its understanding of the disaster area over multiple possibly asynchronous missions. The goal is to improve the team's understanding of how to work in the area and to improve team work between humans and between human and robot. The TRADR use cases involve response to a medium to large scale industrial accident by teams consisting of human rescuers and several robots (both ground and airborne). TRADR missions will ultimately stretch over several days in increasingly

dynamic environments [1].

The research discussed in this paper is closely related to the TRADR-project in that its research should be of interest for, and directly applicable in the TRADR framework. Because of this, the Human-Robot Teaming (HRT) project was created at TNO[2] (Soesterberg, The Netherlands) and this research paper is written in concordance with this project. The Human-Robot Teaming project is meant to be a separate project with its own distinctive goals, but provides support for the TRADR-project at the same time. Where TRADR is focussed on disaster response areas where (typically) the fire department plays an active role, HRT has its roots in militaristic operations and house search missions in particular.

1.2 The Need for Adaptive Behavior

When a robot participates as an active team member in combination with humans in a militaristic house search mission such as in the HRT-project, it will need to operate in complex and dynamic situations that could change any moment. In order to operate functionally well, a robot needs to be able to adapt its behavior depending on the situation. Adaptive behavior is a type of behavior that is used to adjust to another type of behavior or situation. This is used to characterize the way an individual (or a robot in our case) can shift from a less effective behavior to a more effective, constructive behavior, allowing for a more positive outcome, depending on the effect it tends to improve.

The complexity of the behavioral model of the robot correlates with the complexity of the environment. A more complex environment has more variables influencing the effectiveness of the strategy chosen by a robot. A vacuum cleaning robot in a controlled lab experiment in a square room of 10 square meters, with no obstacles and easy-to-recognize dirt to clean up, may perform very well (always recognizing the dirt and always cleaning it up). Once the same vacuum cleaning robot is used in a real-world scenario, things become more complex. Chairs, tables, many corners, stairs and difficult to recognize dirt in difficult to reach positions may greatly influence the performance of the robot, requiring a different skill set and a different approach. It will, for example, need

[1] http://www.tradr-project.eu
[2] https://www.tno.nl/nl/over-tno/locaties/?q=&cat=&gsa_City=Soesterberg

to make sure to keep away from the stairs. Falling down may damage the (most-likely expensive) robot.

1.3 The Need for Simplification

Robots don't think or communicate in natural languages like we humans do. They are operated by a programming language, a formal language that generally consists of instructions for a computer. A more complex environment requires more and different types of behaviors, resulting in a more complex robot with more code. Writing the code on which robots function requires technical skills and is limited to those who are familiar with the system of the robot.

When a robot executes adaptive behavior in ever-changing environments, it should have a vast repertoire of different behaviors. A behavior can vary depending on the situation at hand. It may need to execute the behavior "open door" when standing at a door, or execute the behavior "report dangerous object" when it finds an object worth reporting such as blood stains or weapons. These behaviors have to be created in order for the robot to know how to react in certain scenarios.

Easily modifying and creating the behaviors of the robot as a human operator would greatly benefit human-robot teaming. Robots in the HRT-project are active in high-intensity domains where military soldiers are accompanied by an unmanned ground-vehicle (the robot) on a house search mission. If the robot needs to adapt swiftly to an abrupt change in the scenario, the robot should have the proper behavior that corresponds to this abrupt change. Easily-creatable behavior should not require the need for technical expertise to change and/or write code on which the robot functions.

1.4 Problem Description and Research Question

A more complex environment requires a more complex robot that is able to execute more and different types of adaptive behavior. The robot needs to apply different strategies to cope with the current situation that is happening in real time. If a robot is exploring a house and finds an explosive, it should be able to do more than just "explore" and (at the very least) communicate

the location of the explosive to its human supervisors. If the robot only knows how to explore, it may accidentally trigger the explosive, or miss the fact that an explosive is actually dangerous and should be handled in some way. It is clear that a robot in such an environment should have a vast repertoire of behaviors and strategies in order to function well, because mishandling a situation or not having the proper behavior for a situation could result in deathly casualties.

Implementing more and different behaviors requires technical knowledge, time and resources. Simplifying the process of modification and creation of robotic behavior by the human operator could greatly improve several aspects of the system on which the robot functions. Firstly, the system will be more *accessible*. More people will be able to understand the system and participate in understanding/creating behavior. The professional programmer is no longer the only one able to access the system. Secondly, it will be easier to *modify* the behavior by either changing existing behaviors or adding new ones, attuned to the specific situation. Thirdly, it will be *cost-reducing* because of the improved accessibility by different users (with less technical programming skills).

Because of this, the research discussed in this paper aims to simplify the creation and modification of adaptive behavior for a robot. The following research question, divided in two sub-questions, is formulated:

Research question:

1. How can a framework for robots be created that simplifies the off-line creation and modification of adaptive behavior?
   (a) How can this technically be realized?
   (b) How can this be operationally embedded?

With off-line creation, it is meant that the creation of behavior is done before the actual mission is put in motion. When robots roll out of the factory, they can still be regarded as having a clean slate with no pre-configured behaviors. With the off-line creation of behavior and a framework that can be easily implemented on newly assembled robots, these robots can be provided with a basic model of behavior strategies that can be tweaked and altered depending on the preferred usage of the robot.

Within the technical realization, two questions are central. *When* will certain behavior be performed? *Who is in charge* of choosing this behavior strategy? In order to give answers to these questions, we chose two methods and their integration to tackle these problems. To determine *who is in charge* for choosing a specific behavior strategy, a policy-based approach is used. A policy is a deliberate system of principles to guide decisions and achieve rational outcomes (Kagal, Finin, & Joshi, 2003). A policy is a statement of intent and is implemented as a procedure or protocol and are based on prohibitions and obligations. Policies will be used to determine if a robot has to follow the commands of the human supervisor, or is able to freely choose its own strategy, disobeying the strategy designated by the human operator. Some situations may require the need for quick decision making with an appropriate, pre-defined reaction. Consider the self-driving car example, where a kid crossing the street will result in an immediate activation of the breaking system by the operating system of the car. Even if the human driver may want to drive through (because the driver didn't notice the kid), the car will ignore this and decide that breaking is the best option - and should be followed considering the situation. This way, policies are used to determine who is in charge of choosing a behavior strategy - human, or robot. The policies used in our research are written in Drools [3], a Business Rules Management System (BRMS).

The policies combined can be seen as a system of rules, and a way is needed to translate these rules into actual behavior. If the robot has the policy to take a photo of every painting in the house, it should also have the behavior 'take photo' when a painting is visible for the camera. Behavior trees will allow the actual implementation of these *intentions* into *actions*. Behavior trees will be used for *when* certain actions will be performed. A behavior tree is a hierarchical tree with nodes that control the flow of decision making of an AI (Artificial Intelligent) entity (Marzinotto, Colledanchise, Smith, & Ögren, 2014). At the end of the tree are the leaves. They are the actual actions that control the AI entity. The branches of the tree are different kind of utility nodes that control how the tree will be walked down to the leaves. A utility node (a branch of the tree) may restrict the firing of the leaves (the actions) till a certain condition is met. As an example, the action node *open door* may only be triggered once the condition *standing in front of door* is met. A behavior tree can be extremely deep consisting of many sub-trees where a sub-tree represents a specific behavior or strategy.

[3] `http://www.drools.org/`

The research question regarding how this framework can be operationally embedded will be answered by performing a case study. A house search mission has been simulated in a virtual environment. A script was written that resembles a real-world situation and has been developed with domain experts that work in close collaboration with TNO. An evaluation is given based on the application of the framework in this case study.

1.5 Scope of this Thesis

The objective of this thesis is to design a framework to enhance human-robot teaming by simplifying the creation and alteration of adaptive robotic behavior. The framework consists of a policy-driven behavior tree that is implemented in a virtual simulated environment, created in Unity [4]. The policies are written in Drools [5] and the behavior tree is created in Behavior Designer, a third-party behavior tree implementation plugin for Unity [6]. The policies and the behavior tree are created separately. Because the framework should be accessible to more users without the need to actually change code, a graphical user interface is used for the implementation of behaviors in the behavior tree and policies in Drools. In collaboration with TNO [7] a tool is developed that is capable of linking policies to sub-trees within the main behavior tree. This way, policies translate directly to behavior performed by the robot. The tool developed by TNO is called PET (Policy-Engine TNO) and shows which policies are active and which policies are not, while at the same time allowing the human operator to enable or disable specific policies.

This will be researched by performing a case study. A militaristic house search mission has been chosen to demonstrate a scenario that requires adaptable behavior. A script was written that allows a human operator to play, or operate, the virtual simulation in a 3D environment with the robot as a team member, accompanying the human operator during this mission. This script outlines the movement of the robot through the house in the most realistic way, resembling a real-life mission. The script was written in concordance with domain experts that are collaborating with TNO. Ide-

[4] http://www.unity3d.com/
[5] http://www.drools.org/
[6] http://www.opsive.com/assets/BehaviorDesigner/
[7] https://www.tno.nl/

ally, the realization of this framework should be a workable example that can be used in future militaristic scenarios. This virtual environment can be used to gain insights in implications of these robots on current and future operating procedures. To the best of our knowledge, the combination of policies and behavior trees is a novelty and due to its ease of use, could encourage the community to further explore the possibilities of this policy-driven behavior tree combination.

1.6 Thesis Outline

The literature study will be described in Chapter 2. The literature study will be performed to support the choice for policies and behavior trees. How policies work and the integration of policies in Drools will be explained in Chapter 2.1. A deeper, more precise elaboration on behavior trees is given in Chapter 2.2.

After the literature study, Chapter 3 explains the scenario that is used for the house search. The house search scenario is developed in Unity. The choice for a virtual test environment is explained in Chapter 3.1. A map of the ground floor and first floor is given in Chapter 3.2 with routes that the robot will walk.

In Chapter 4 the implementation of policies and behavior trees is explained. It will be explained how this policy-based behavior tree influences the strategy the robot chooses. A more extensive elaboration will be given on how we realized this technically, by explaining policies in detail in Chapter 4.1 and behavior trees in Chapter 4.2. The combination of the policies and the behavior trees, and the policy engine developed by TNO is discussed in Chapter 4.3 and a simulation run is shown in Chapter 5. An answer to the research question will be given in Chapter 6, followed by a discussion.

2 Background literature

As discussed earlier, there are two main questions regarding adaptive behavior. *When* will certain behavior be performed? *Who is in charge* of choosing this behavior strategy? A policy-based approach will be used to answer the question who is in charge for choosing a specific behavior strategy. Policies are introduced in Chapter 2.1. Policies are written in Drools, a Business Rules Management System (BRMS) solution. Whereas the policy-based approach has a strong foundation in creating rules and providing statements of intent, behavior trees will allow the actual implementation of these intentions. Behavior trees will be used for *when* a certain behavior actions will be performed and are discussed in Chapter 2.2. Chapter 2.3 explains why behavior trees and policies are chosen to answer the research question in this thesis.

2.1 Introduction to Policies

A policy is a system of principles with the goal to guide decisions and achieve desirable outcomes. Policies are known to be extensively used in the security, network routing and management systems. Policies have a dynamic nature, without the need to change the internal mechanism of the entities involved (Kagal et al., 2003).

A distributed system is a piece of software that ensures that a collection of individual, independent computers appears to its users as a single coherent system. The need for policies to specify and govern the behavior of distributed systems emerged as a result to answer the increasing complexity within these distributed systems. In order to function in an environment with changing requirements, large distributed systems must be able to change and adapt its behavior while they are still active and running (Bradshaw & Montanari, 2014). An example of a distributed system is a network system. Network management software typically specifies and measures acceptable performance thresholds for each machine in the network without creating additional traffic. It manages and automates administrative tasks across multiple machines in a network, and one method of detecting problems is via a policy. A policy can be seen as a set of specifications that define specific thresholds and conditions that have to be met in order to trigger certain network administrative actions. These policies are constantly searching for abnormal behavior within the network system,

monitoring the system and providing correct actions to prevent server downtime (Richardson, 2001).

In this research, policies are used as a way to easily create and modify adaptable behavior in a robot. We narrow it down to the following definition to avoid any unwanted confusions and to improve the readers' understanding.

> *Policies are a way of defining the behavior of a robot by rules for permissions and obligations.*

In order for robots to understand policies, we will need a suitable policy (or rule) engine. A policy engine is a software component that allows the organization, creation and the ability to monitor and enforce rules. In a policy engine, policies are specified in a machine readable policy-language. The goal is that the policies can be used by a policy enforcement mechanism which runs on every robot and other autonomous system to ensure that each autonomous system at least adheres to a specified set of obligations and permissions. For example, a policy might state that any information about victims must be immediately communicated to the team leader, or a policy might state that robots are not allowed to use their weapons without permission of the platoon commander. We will use *Drools* for our policy engine and the specification of these policies [8] (Salatino, De Maio, & Aliverti, 2016).

2.1.1 Policy Engine: Drools

Knowledge representation and reasoning (KR) is the area of Artificial Intelligence that focuses on how knowledge is represented and manipulated. Knowledge representation is dedicated to presenting information about the world in a form that a computer can understand. Expert systems use knowledge representation for the codification of knowledge into a knowledge database. This knowledge database can then be used for reasoning, meaning that we can process data with this knowledge database in order to deduce conclusions. Drools is a Rule Engine that uses the rule-based approach to implement an expert system and is more correctly classified as a production rule system (Salatino et al., 2016).

[8] `https://www.drools.org/`

A Rule Engine allows you to define *what* to do, and not *how* to do it. Rules are pieces of knowledge and are often expressed in the form `when` a certain condition is met, `then` do some tasks.

```
when
    <conditions>
then
    <actions>;
```

Imagine this simple scenario. When a robot is near a door, we want the robot to open the door. The Drools rule implementation for this could be the following.

```
rule "OpenDoor"
    when
        Robot (location == Location.Door)
    then
        action(OpenDoor);
end
```

The `rule "OpenDoor"` states that when the robot is at the location of a door (specified by the code `Robot (location == Location.Door)`), then the robot will open the door (specified by the action `action(OpenDoor)`).

The brain of a production rule system is an inference engine that is able to process a large number of rules and facts. The process of matching the new or existing facts against production rules is called pattern matching, which is performed by the inference engine. The inference engine matches facts and data against rules to infer conclusions which result in actions (Salatino et al., 2016). Drools uses the Rete algorithm for the inference engine. When a large number of rules and facts are used, multiple rules may be applicable for the same facts assertion, resulting in rules that are in conflict. When this is the case, the agenda will define the order of execution of these rules. Chapter 4.1.2 discusses how conflicting policies are handled in our case study. The Rete algorithm is used to determine which of the system's rules should fire based on its data store. A detailed treatment of

Rete is beyond the scope of this thesis. For more information about this algorithm, see the original article (Forgy, 1982).

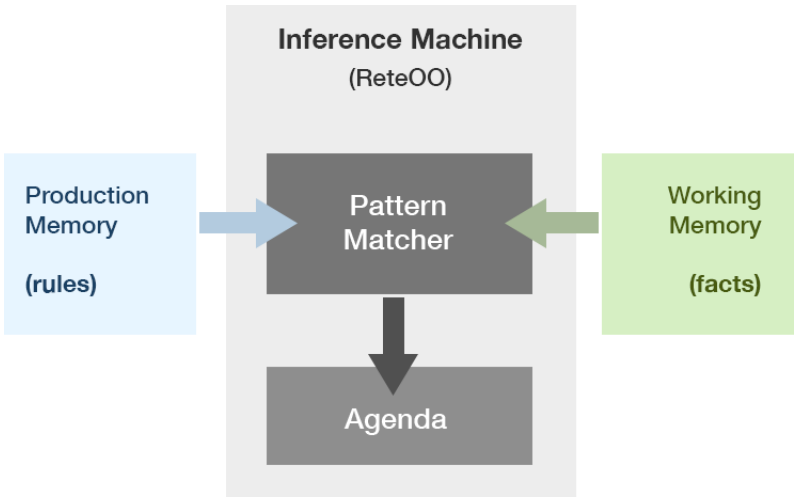


Figure 1: High-level view of a Rule Engine

Figure 1 illustrates the functioning of a rule engine. Rules are stored in the production memory and the facts are stored in the working memory. The pattern matcher matches the rules against the facts. To illustrate this with a working example, take the `rule "OpenDoor"` we created earlier. The complete rule is stored in the production memory. The pattern matcher will check if the conditions for this rule are satisfied. So if the fact `Robot (location == Location.Door)` is `true`, the working memory will return that this condition (or fact) is true. The pattern matcher will match this rule and the fact, pushing it to the agenda, leading to the action `action(OpenDoor)`.

2.2 Introduction to Behavior Trees

A Behavior Tree (BT) is used to organize the switching between tasks of an autonomous agent, such as a robot or a virtual entity in a video game. BTs were developed in the game industry and later adopted by the academic world (Marzinotto et al., 2014). In the game industry, BTs functioned as a powerful tool to shape the behavior of NPCs (non-player characters). BTs excel at creating complex systems that are modular and reactive, properties that are crucial for many modern applications (Colledanchise & Ögren, 2017). Reactivity is the ability of the robot to swiftly react to changes. If a robot is about to collide with a wall, it should be able to quickly steer away from the wall, avoiding collision. Modularity means that certain behaviors or parts of behaviors can be seperated into building blocks and recombined in others (Gershenson, Prasad, & Zhang, 2003). When the robot becomes modular, components of the robot can be enabled, tested, created and reused independently of one another. Because of this modularity and reactivity, the application of BTs has been extended beyond video games towards AI and robotics. More recent works propose the use of behavior trees for robotic manipulation, multi-robot systems and multi-mission control frameworks for UGVs (Unmanned Ground Vehicles) and UAVs (Unmanned Air Vehicles, such as a drones) (Marzinotto et al., 2014)(Colledanchise, Marzinotto, & Ögren, 2014)(Ogren, 2012)(Klöckner, 2013).

Chapter 2.2.1 describes Finite-State-Machines (FSMs), the predecessor of BTs. In games, the control structures of NPCs were often defined by FSMs before BTs were developed. In Chapter 2.2.2 BTs are introduced. BTs provide an alternative option for FSMs because they support modularity. Sub-trees within the main tree can express a particular behavior, and these sub-trees can be re-used in other parts of the main tree.

2.2.1 Finite-State Machines (FSMs)

BTs originated from finite-state machines (FSMs) (Colledanchise & Ögren, 2017). A FSM is a technique that can be used to implement the behavior of agents and can be seen as a model where each state is a behavior that can be executed by the agent. The FSM consists of a set of states, transitions and events. See Figure 2 for an example FSM, designed to carry out a simple explosive-search task. Only one state can be active at a time and is defined as the current state. The

current state determines the possible next action the agent can make, changing the state from one to another. For example, the robot may be in state `Patrol`, from which it will enter the state `Find explosive` if and only if an explosive is near, as illustrated by the transition `Explosive is near`. If no explosive is near (as illustrated by the transition `Explosive is not near`), it will continue patrolling. Once it enters the new state, it will be limited to the choices given in this new state. So once an explosive is near, the state `Find explosive` becomes active. The robot will then communicate its location, resulting in the active state `Report explosive`. When communication is successful, the robot will go back to patrolling the area.

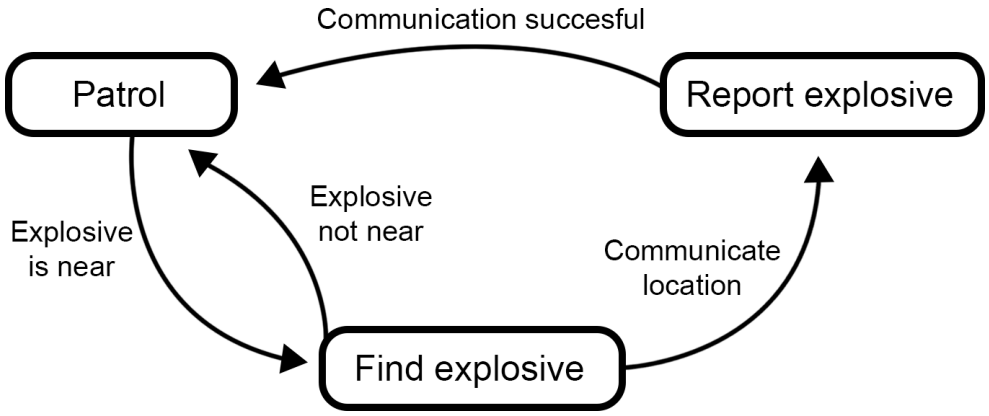


Figure 2: Graphical representation of a finite-state-machine designed to carry out a simple explosive-search task, showing its possible states (squares) and transitions (arrows).

FSMs are used because of three main advantages (Colledanchise & Ögren, 2017).

- Easy implementation
- Easy to understand/intuitive use
- It has a very common structure

There are, however, some problems with FSMs in real applications when the system modelled grows in complexity with more states (Colledanchise & Ögren, 2017).

- **Scalability.** Imagine a model with 500 states instead of 3 as in Figure 2. Removing or adding one state could lead to complex problems that are difficult to solve, because this new state may be linked to a couple of hundred other states, changing the whole model. A FSM with

many states is difficult to modify by both humans and computers.

- **Maintainability.** When there are 500 states, the graphical representation of the model becomes hard to read and understand due to the number of states and transitions connecting states. Adding or removing a state in the model requires the re-evaluation of practically every transition and internal state in the model. Because of this, FSMs are highly susceptible to human error.
- **Reusability.** Because the states are strongly connected and interdependent of each other, it is difficult to copy a certain behavior from one model to another.
- **Parallelization.** It is difficult to run multiple states parallel in one FSM model, often requiring external modules to resolve conflicts and deadlocks.
- **Goal-oriented.** The model knows when a state is active, but it is independent of the previous states. This means it will be difficult to coordinate states to achieve a certain goal.

2.2.2 What are Behavior Trees?

A behavior tree (BT) is a directed rooted tree where the leaf nodes are called execution nodes (the actual commands that control the robot) and the internal nodes are called the control flow nodes (Thulasiraman & Swamy, 1992). There are two types of execution nodes (action and condition) and four types of control flow nodes (sequence, fallback, parallel and decorator). These node types are discussed below and summarized in Table 1. A node can be a parent or a child in relation to another node. The root of a BT (at the top) is the node without any parents; all other nodes have exactly one parent. The control flow nodes have at least one child (Colledanchise & Ögren, 2017).

A node can have 4 different states (Champanard, 2012). These are:

- SUCCESS : The success status will be returned when a condition node has met a certain criteria, or when an action node has been successfully fired.
- FAILURE : The failure status will be returned when a condition node has *not* met a certain criteria, or when the action node could not be fired due to a particular reason.

- `RUNNING`: The running status will be active when a node is initialized but still waiting to be completed.
- `ERROR`: The error status will be returned when the node could not be executed due to some error, probably caused by a programming error.

Action nodes. Action nodes are most likely the easiest to understand in the way that they alter the state of the game in some way. The action implementation depends on the agent type. For example, the actions of a robot may be defined by making a sound through the speakers or making a photo with the camera. For an NPC (non-player character in a digital simulation) the action could be directed towards playing a certain animation or a sound. Actions don't have to be external (interacting with the environment), but could be internal too (creating a registry log, saving a file and so on). An action node will return `SUCCESS` if the action has been completed, `FAILURE` if the action was not completed and `RUNNING` while still executing the action.

Conditional nodes. A conditional node tests whether a certain condition is met or not. The conditional node must have some target variable, such as `target distance` and a criterion to base the condition (`target distance` < 100). These nodes will return `SUCCESS` if the condition has been met or `FAILURE` is the conditional node was not met.

Sequence nodes. The sequence node controls its children from the left until it finds a child that returns either `FAILURE` or `RUNNING`, then it returns `FAILURE` or `RUNNING` accordingly to its own parent. It returns `SUCCESS` if and only if all its children succeed. Note that when a child returns running or failure, the sequence node will not evaluate the state of the next child (if there are any). The symbol of the sequence node is a box containing the label $\rightarrow$, shown in Table 1.

Fallback nodes. A fallback node measures its children from the left until it will find a child that either returns `SUCCESS` or `RUNNING`. When it has found such a child, it will return the status of its child to its parent. So if the child returns `RUNNING`, the status of the fallback node will be reported as `RUNNING`. It returns `FAILURE` if and only if all of its children return `FAILURE`. A fallback node only needs one child to return `SUCCESS` to succeed himself, in contrast to a sequence

node which requires all its children to succeed. If one child returns RUNNING and another child returns SUCCESS, the fallback node will still return SUCCESS.

Decorator nodes. The decorator node is a control flow node with a single child that manipulates the return status of its child according to a user-defined rule. For example, an invert decorator inverts the success/failure status of the child, so if the child may return FAILURE, the decorator node will return SUCCESS by inverting the status of its child (Colledanchise & Ögren, 2017).

Node type	**Symbol**	**Succeeds**	**Fails**	**Running**
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Sequence	$\rightarrow$	If all children succeed	If one child fails	If one child returns running
Fallback	?	If one child succeeds	If all children fail	If one child returns running
Decorator	$\diamond$	Custom	Custom	Custom

Table 1: All the node types for behavior trees (Colledanchise & Ögren, 2017).

Figure 3 shows a simple high level BT of finding a painting, taking a picture and sending the picture.

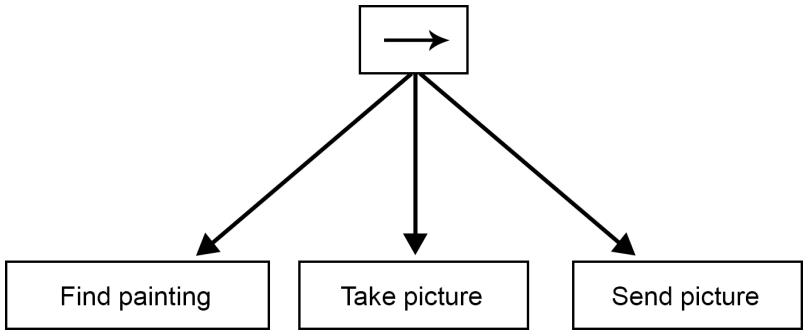


Figure 3: Graphical representation of a behavior tree, designed to carry out the simple node to find a painting, take a picture of the painting and send the picture.

The root of the BT in Figure 3 is illustrated with the control flow node $\rightarrow$ (a sequence node type) and has three children, all action nodes. A BT is always read from left to right. When Find painting

is returned with SUCCESS, the second action Take picture is RUNNING. Once its status is changed from RUNNING to SUCCESS, the third action Send picture will be RUNNING. Once the status of the last action node in this sequence is successful, all children in Figure 3 have returned successful, so the complete sequence returns SUCCESS (see Table 1; a sequence succeeds if all children succeed).

A BT can be extremely deep, where certain nodes contain sub-trees that specify a certain behavior. With these sub-trees, the developer can create and link many different behaviors together, resulting in rich, adaptive behavior. An example of a sub-tree is given in Figure 4.

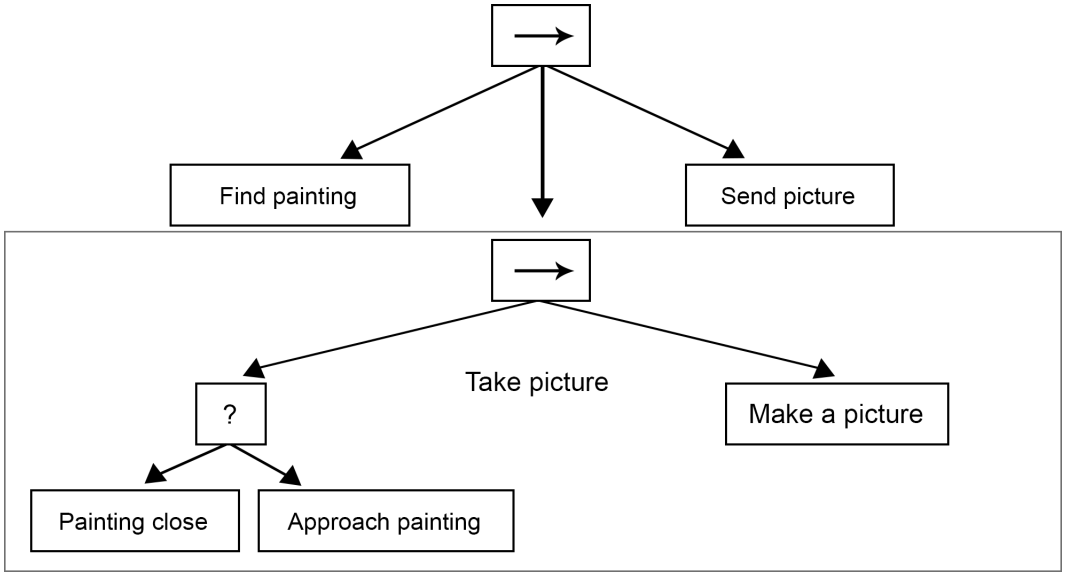


Figure 4: The action "Take picture" from Figure 3 is accommodated in a sub-tree, where the painting is approached until close enough, resulting in the execution of the camera taking a picture.

This sub-tree, captured in the node Take picture, contains a sequence node → with one fallback node ?, and this fallback node contain two execution nodes. As Table 1 points out, the sequence node at the top is successful if all its children succeed, so it will need both the fallback node and the action node Make a picture to return SUCCESS. The fallback node succeeds if one of its children succeeds. So, if it finds a painting, it will measure if the painting is close enough. If this is not the case, it will evaluate the action node Approach painting and if this returns SUCCESS, the fallback node ? will return SUCCESS, continuing with the action node Make a picture. After this, the

complete sequence returns `SUCCESS`, evaluating the `Send Picture` node. Once this node returns `SUCCESS`, the whole tree displayed in Figure 4 returns `SUCCESS`.

2.3 Summary: Policies and Behavior Trees

BTs have been used, and are still used, extensively in the field of robotic manipulation, in contrast to FSMs. Modularity is one of the main advantages of BTs, because (parts of) individual behaviors can be reused in different applications and behaviors, without needing to specify how these newly imported behaviors relate to subsequent behaviors (Bagnell et al., 2012). BTs are simple to design and implement, scalable when models become larger and more complex, and support modularity to create reusability and portability. One of the main advantages of using behavior trees is that they are easy to understand and can be created using a visual editor, giving end-users the power to create adaptive behavior with the same amount of complexity as traditionally-written programs (Paxton, Hundt, Jonathan, Guerin, & Hager, 2017). Because of this, implementing, using and changing behavior trees is less error prone.

Policies define *what* to do with obligations and permissions, and the behavior trees define *how* to do it. By using an ontology, it is possible to give the robot a sense of understanding about its surroundings, allowing it to make decisions on its own. An ontology can be seen as an expression of a terminology, allowing the system to classify basic objects, statuses and actions and how they may relate to one another. For example, a `Robot` is classified as an `Actor` and has the status `Location` which tracks the location of the robot. Because of the modularity of BTs and their ease of implementation, it is possible to combine policies with behavior trees. As discussed, a behavior tree can become very big, with many sub-trees controlling the behavior of the robot. When policies are attached to sub-trees of the original behavior tree, the behavior of the robot can be controlled by *disabling* or *enabling* these sub-trees. For example, take the following policy: `Robot is NOT allowed to move`. When linking this policy to nodes in the behavior tree that define movement, these movement nodes will return `FAILURE` on firing this node since movement is not allowed, defined by the active policy. This means that the particular part of the behavior tree is not executable. How this was realized technically can be found in Chapter 4.

3 Scenario for Adaptive Behavior

This research focusses on robots as autonomous team members in a team combined with humans, instead of the alternative where robots are operated by a human. In order to answer the research question on how to develop a framework that allows the creation and alteration of behavior, it has been decided to go for a militaristic house search scenario. In such a mission, a large variety of factors may influence the required behavior of the robot in order to operate effectively. Professional teams working in safety-critical domains (such as in the military domain) often operate in unpredictable, stressful, time-pressured environments. Failure or malfunction may result in death, loss of equipment/property or severe damage. The robot will need to be adaptive in such an environment while scanning the area for guns and other dangerous objects.

The collaboration between the human and the robot will be in an operational task. This task should not be too complex and the usage of a robot should show enough realism. To successfully participate as a complete team member, the robot should possess a diverse set of communication skills, e.g. to whom it should report its findings or if it is allowed to pick up objects without the permission of the human team member. There should be a possibility for the robot to communicate with the human team member, allowing the behavior of the robot to change based on the strategy provided by the human team member through several available options.

Creating a house search mission in real life is an expensive and time-consuming process. For this master thesis, there were no funds available to sponsor such a scenario. Because of these limitations, a more practical approach was taken that has some benefits over a real world scenario. In order to answer the research question discussed in Chapter 1.4, a virtual environment was created. Chapter 3.1 indicates the choice for a virtual environment and why it suits this case study. Chapter 3.2 shows the floor map of the house in a top view. The locations of dangerous objects are marked and the overall route of the robot from start to finish is explained. Chapter 3.3 explains the scenario and discusses the objective of the house search mission. The script was written in concordance with domain experts working closely with TNO and should reflect a realistic scenario that is based on actual field work.

3.1 Virtual Environment

Digital games and robotics in real-life scenarios have a common goal. That goal is to design smart, autonomously behaving and adaptive agents that are able to interact with the environment (whether a real environment or a digital simulation) and to interact with other agents (whether those are humans or not). An applied game has the intention to express a 'serious' note that often entails the capability to learn or experiment with something new in a meaningful, practical, yet virtual way (Peeters, Van Den Bosch, Meyer, & Neerincx, 2012). These applications are often found in areas like scientific exploration, health care, education, defense and others. They often encompass some form of simulation, which is the imitation of the operation of real-world processes over time (Susi, Johannesson, & Backlund, 2007). By creating a virtual environment that is suited for testing policy-driven behavior trees instead of building a real-world test environment, a quicker, less expensive and more controlled environment can be created. And maybe the most important argument is that building a virtual environment instead of a real world test environment, is equally (or better) suited to answer the research question: *How can a framework for robots be created that simplifies the off-line creation and modification of adaptive behavior?* A virtual environment is a more controlled environment that is easier for testing and development.

The virtual environment has several **requirements** in order to answer the research question:

- A realistic 3D representation of a house.
- The possibility to implement an agent in the virtual environment (the robot exploring the house).
- The possibility for a human to operate the scenario in collaboration with the robot.
- The possibility to coordinate movement and behavior of the robot through behavior trees.
- The possibility to control parts of the behavior tree through policies.

Unity was chosen as the game engine to build this virtual environment [9]. Unity is a powerful cross-platform, all-purpose game engine that is not only used for developing games, but also suited for developing simulations and scenarios such as the one discussed in this research. Unity allows for

[9] `https://www.unity3d.com/`

the implementation of agents and allows for a human counterpart to collaborate together with the robot on the mission. Unity has an Asset Store that is home to thousands of assets, developed by third-party developers [10]. These assets are free or accessible for a small fee. One of these assets is Behavior Designer, developed by Opsive [11]. Behavior Designer is a behavior tree implementation with an intuitive visual editor that works well together with Unity. More information about the technical implementation of the behavior trees and how they work can be found in Chapter 4.

3.2 Map of the House

With Unity, it is possible to build a realistic 3D house that is suited for a house search mission. For our research, an existing Unity Asset was found with a 3D model of a house [12]. This house was further altered in order to suit our needs for the mission.

[10] https://www.assetstore.unity3d.com
[11] http://www.opsive.com/assets/BehaviorDesigner/
[12] https://www.assetstore.unity3d.com/en/#!/content/48976

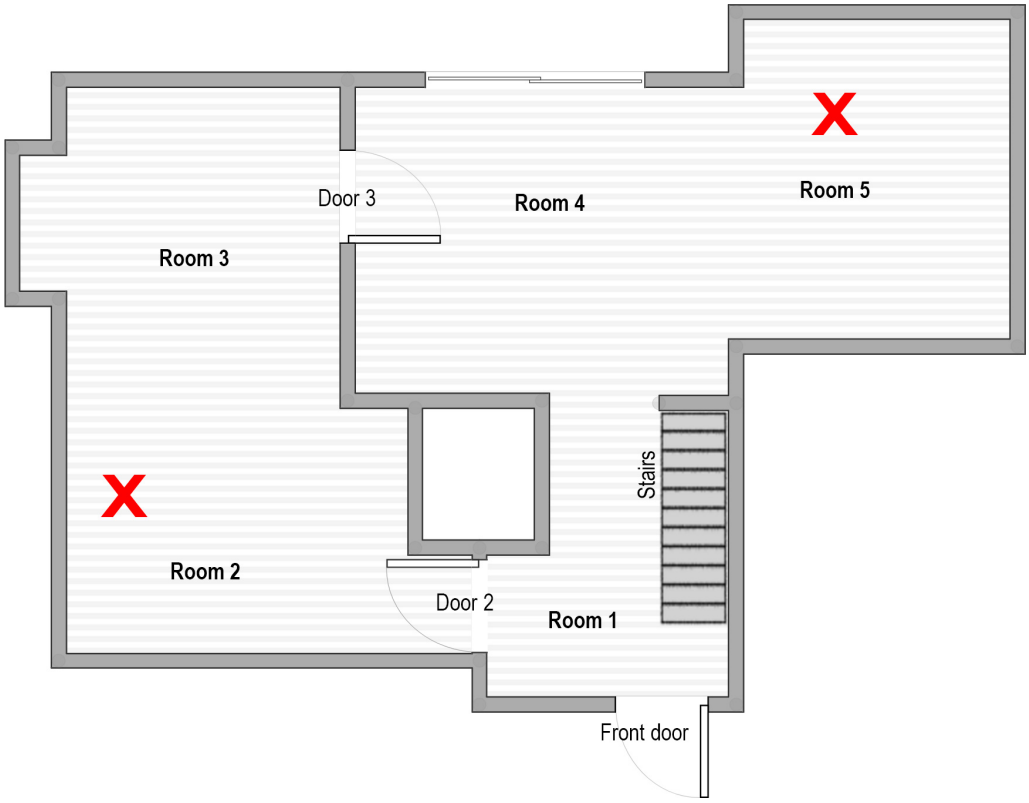


Figure 5: High-level view of the map and the route of the robot - floor plan first floor.

Figure 5 shows the floor plan of the first floor in the house. This map was created using an online house-planning application [13] and is an identical match of the virtual 3D house in Unity. There are no people in the house. We assume there are several objects of interest in the house. The robot will enter the house through the front door. A script was written that outlines the route of the robot and some of its points of interest. The script was written to resemble a house search mission, but then altered for use with a robot. The script was written and tested in concordance with domain experts working with TNO [14]. Several weapons are placed through the house and are marked on the floor plan with a big red X. Once the robot finds a weapon, it will open a dialogue with the human operator. The human operator will be able to press a button to order the robot to pick up the weapon. There are also other objects that can be found randomly in the scene. These objects

[13] `https://www.homestyler.com/floorplan/`
[14] `http://www.tno.nl/`

are boobytraps, a fire, or explosives. How these situations are interpreted by the robot is discussed in detail in Chapter 4.1 and Chapter 4.3. The robot will move through the rooms, starting with *Room 1*, till it reaches *Room 5*. Then the robot will move towards the stairs and make its way up to the second floor, illustrated in Figure 6.

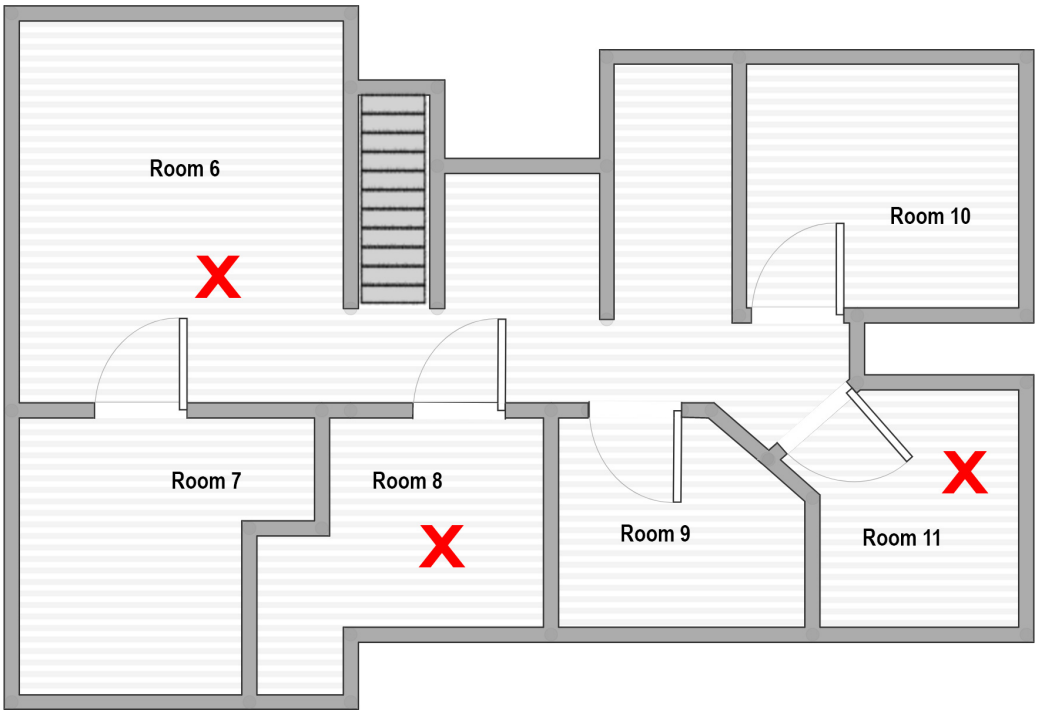


Figure 6: High-level view of the map and the route of the robot - floor plan second floor.

After the robot finishes searching the ground floor, it will use the stairs to move to the second floor. Figure 6 shows a high-level view of the second floor and the locations of weapons, marked by a red *X*. The same procedure will be used as on the ground floor; when the robot finds a weapon or other dangerous objects, it will activate a specific behavior strategy corresponding to the object it finds. The robot will move through the rooms, starting with *Room 6*, till it reaches *Room 11*. The scenario will end when the robot reaches *Room 11*.

3.3 Scenario

As discussed in Chapter 3.2, the robot moves from room to room, searching for guns or other dangerous objects found throughout the scenario. The scenario in Unity is played as a soldier that accompanies the robot and works just like any other 3D video game. Looking around can be done by moving the mouse and the *WASD* keys on the keyboard function as movement keys. Interaction with objects (such as doors, light switches and other objects) is possible with an interaction button, mapped to the *E* button on the keyboard. The robot can be out of sight from the player and moves autonomously. The human operator is allowed to move freely through the house. An image of the robot is shown in Figure 7.

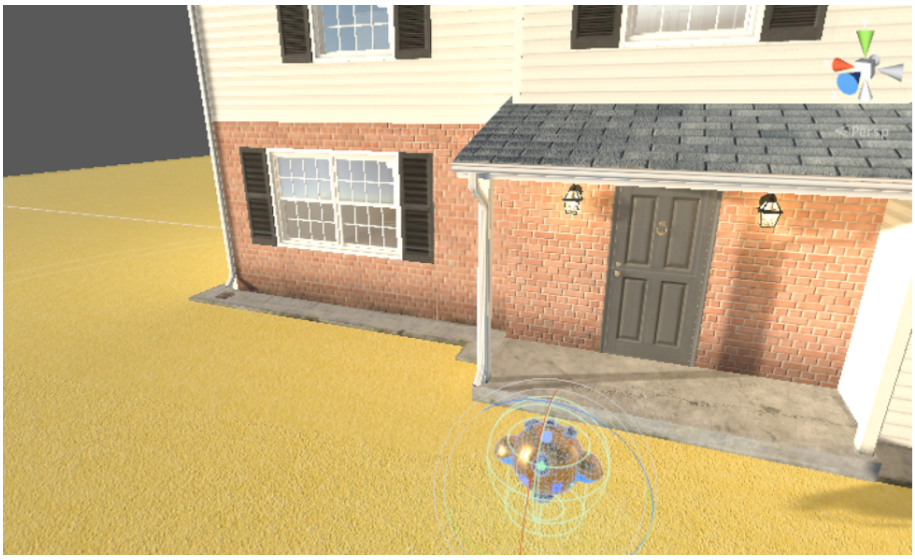


Figure 7: A preview of the robot that accompanies the human operator.

Figure 8 illustrates the situation inside the house, standing on the ground floor just before the staircase (see Figure 5 for the exact location).

Figure 8: Example of what the house looks like.

Figure 9: Example of a gun lying on the floor.

Figure 9 gives an example of a dangerous object (in this case a weapon), placed somewhere in the house. The moment the robot sees a weapon, it will move towards the weapon and a dialogue will be started between the robot and the human operator. The scenario will be paused and the robot

will stop moving. In the particular case of the weapon encounter, the human operator will be able to order the robot to pick up the weapon. The robot will then pick up the weapon and it will be removed from the actual scene. The robot will then continue the house search, moving to the next room in line. The scenario will end when the robot reaches *Room 11*, the last room in the house.

The description of the scenario given here is rather limited and functions mostly as a stepping stone for the next part, where we explain the technical details and how the policies and behavior trees connect and work together. Chapter 5 gives a more detailed step-by-step explanation of a simulation run and is an extension of this Chapter.

4 Policy-Driven Behavior Trees

In this Chapter the combination of policies and behavior trees will be explained. But before this combination will be discussed in Chapter 4.3, a deeper elaboration will be given in Chapter 4.1 how policies work and which policies are used. A complete list of all the policies used in the scenario is given in Chapter 4.1.3. The actual behavior tree that has been used will be discussed in Chapter 4.2, describing several specific behaviors in detail.

4.1 Policies

Chapter 2.1 already gave a general description of policies and how they work in Drools. This subchapter gives a more in-depth explanation of how policies are used in the scenario described in Chapter 3. First, the ontology is introduced and described in Chapter 4.1.1, allowing the system to classify basic objects, statuses and actions and how they may relate to one another. For example, a `Robot` is classified as an `Actor` and has the status `Location` which tracks the location of the robot.

After the ontology, deontic logic is introduced in chapter 4.1.2. Deontic logic is used to define the logical construction of our policies based on prohibitions and obligations and is used as a formal method of representation for our policies. Here, the policies will be introduced and some alternative policies are discussed that resolve conflicting policies. A complete list of the policies is given in chapter 4.1.3. Chapter 4.1.4 explains how policies are constructed in Drools.

4.1.1 Ontology

An ontology is used to provide a shared and common understandig of a domain that can be communicated between different agents (humans and software). They have been developed in the field of Artificial Intelligence to facilitate knowledge sharing and reusing this knowledge, and is used as a terminology. An ontology organizes information by representing entities, objects and events, along with their properties and relations, according a to a system of categories. An ontology is a description of the concepts and relationships that can exist for agents (Fensel, 2001). Every object is an instance of an ontology class, providing a clear structure for the data that allows software

agents and humans to understand each other. For example, a robot is a child of an agent, or an obligation is a child of a policy decision.

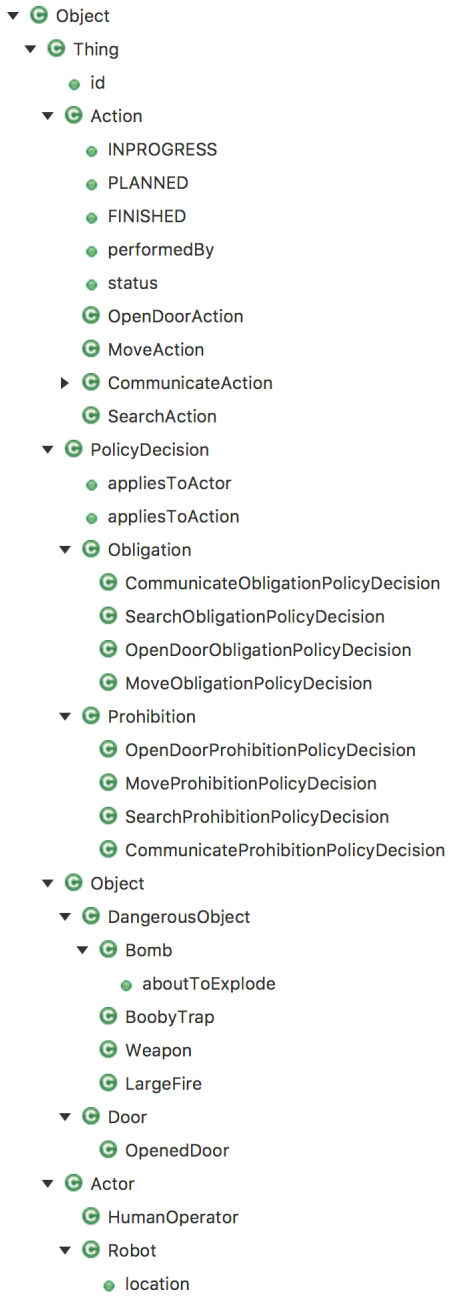


Figure 10: Ontology- tree view

A list of the ontology classes used in this research is shown in Figure 10 and is also included in Appendix C. Here, we see a tree view for all the ontology classes and how they relate to other classes. A `Door` is an `Object` with the available property `Opened Door`. The ontology class `Action` has several properties it may adhere to, such as `In Progress`, `Planned`, `Finished` and others. All the classes are *Java Classes*.

4.1.2 Policies and Logic: a Deontic Concept

Policies, just as legal systems, are based on prohibitions, obligations and permissions. Deontic logic is the logic that is concerned with these concepts and attempts to capture their essential logical features. With deontic logic, it is possible to further define the logical basis of these concepts (Prakken & Sartor, 2015). Policies can be used to allow a user to restrict the robot's autonomy in a dynamic way. Deontic logic uses *Obligations*, defined by **O**, and *Permissions*, defined by **P**. $\mathbf{O}\phi$ means that it is obligated that ϕ, and $\mathbf{P}\phi$ means that it is permitted (or permissible) that ϕ. Negations, denoted by $\neg$, allow the construction of prohibitions: $\mathbf{O}\neg\phi$, meaning that it is obligated to not do ϕ.

There are three basic behaviors (*movement*, *communication* and *searching*), each with an *obligation* rule and a *prohibition* rule. Obligations state that the robot is obligated, or committed, to do the specified behavior. Prohibitions state that the robot is prohibited, or forbidden, to do the specified behavior. These policies are not allowed to be active at the same time, or simply cannot be active at the same time, because they are in conflict with one another. The robot cannot adhere to the policy `ObligateMove`, obligating it to move, while simultaneously applying the policy `ProhibitMove`, prohibiting it from moving. There is either movement, or no movement. Thus, $\mathbf{O}move \wedge \mathbf{O}\neg move$, where the *move* stands for movement, cannot be true. In order for these policies to have some method to resolve these conflicts, an *ontology* is used as a terminology and a way of classifying objects and situations that can be used in the creation of new policies.

Normally, the human operator chooses the active policy and the robot will follow. How the human operator can choose which policies the robot should adhere to is discussed later in Chapter 4.3. In some cases the robot is able to overrule the current active policy with its counterpart when the robot deems that this new approach is essential for the success of the mission, depending on the

situation. The policy ProhibitMove, issued by the human operator, may be overruled by the robot with ObligateMove if the robot follows the policy to move outside of the house due to a specific dangerous situation. The following rules were defined, with the following propositions: [*(A)* a large fire], [*(B)* a bomb about to explode], [*(C)* a boobytrap nearby] and [*(D)* an object classified as *Dangerous*].

The policy ObligateMove can be expressed as $\mathbf{O}move$, where *move* stands for movement. Thus, the robot adheres to the policy that obligates it to move. The policy ProhibitMove can be expressed as $\mathbf{O}\neg move$, since prohibiting the robot to move is considered the same as obligating it to *not* move. If the human operator issues a ProhibitMove policy, the robot may choose to overrule this policy with ObligateMove under these conditions: if there is a [A: large fire], or there is a [B: bomb about to explode], and there is no [C: boobytrap nearby]. The idea here is that a boobytrap might be triggered by moving the robot, causing an explosion that might harm the robot and the mission, and moving away from an exploding bomb or a large fire may result in a successful retrieval of the robot, increasing the success rate of the mission. This corresponds to the logical syntax $(A \vee B) \wedge \neg C \supset \mathbf{O}move$, where $\vee$ is equivalent to the *or* operator, $\wedge$ stands for the *and* operator and *move* means to move outside of the house. Thus, *if* (A) there is bomb about to explode, *or* (B) there is a large fire, *and* (C) there is no boobytrap nearby (that may be triggered by moving) *then* the robot is obligated to move outside of the house. It is believed that these situations require the robot to move to safety. There is an ontology class for a bomb which can have the property *about to explode*, and there is an ontology class that defines a *large fire* and a *boobytrap*. These objects/situations (boobytrap, large fire) or statuses of an object (bomb (object) that is about to explode (status)) can be recognized by the robot in the virtual environment. If these ontology classes in the current scene correspond with this logical rule, the robot will activate the ObligateMove policy with the corresponding behavior and moves outside. The other way around, if an ObligateMove policy is active, the robot can overrule this with ProhibitMove if [*(C)* a boobytrap is nearby]. This can be logically structured with $C \supset \mathbf{O}\neg move$, because moving may trigger the boobytrap from exploding.

The policy ProhibitCommunicate prohibits the robot to communicate with the human operator, $\mathbf{O}\neg comm$, where *comm* stands for communicating with the human operator. This policy can be overruled by the robot with the policy ObligateCommunicate if the robot encounters an object or

situation with the status *Dangerous*, obligating the robot to communicate this to the human operator. As can be seen in Figure 10, the status *Dangerous* is applicable for the objects *weapon*, *bomb*, *large fire* and *boobytrap*. It is believed that if one of these situations hold true in the current environment, it is important to communicate this with the human operator, even it it means to ignore the current `ProhibitCommunicate` policy. Once these objects are within sight of the robot, the robot knows these objects are classified as dangerous and the human operator should know about them. So, if we have the proposition [*(D)* an object with the status *Dangerous*], we have the logical construction $D \supset \mathbf{O}Comm$, where [*(D)* an object with the status *Dangerous*] entails the possible sightings of a weapon, a bomb, a large fire or a boobytrap. We argue that if the human operator issues a `ObligateCommunicate` policy, there are no situations that requires the robot to overrule this policy with its counterpolicy `ProhibitCommunicate`.

The policy `ObligateSearch`, logically structured as $\mathbf{O}search$, can be overruled with `ProhibitSearch` if there is a [*(C)* boobytrap nearby], resulting in the logical structure $C \supset \mathbf{O}\neg search$. It is believed that when a boobytrap is nearby and the human operator may not know about this, it is desirable to stop searching to prevent the boobytrap from being triggered.

4.1.3 Policy List - All Policies

The following policies are all the policies used in the scenario. These are the three basic behaviors *movement*, *communication* and *searching* with an *obligation* policy and a *prohibition* policy as described in Chapter 4.1. Combining these with the exceptions described above in Chapter 4.1.2 we get the following policies, where *(A)* stands for a large fire, *(B)* stands for a bomb about to explode, *(C)* stands for a nearby boobytrap and *(D)* stands for an object or situation classified with the status *Dangerous*. *Move* stands for movement, *comm* stands for communication and *search* stands for searching. *OperatorSaysMove*, *OperatorSaysCommunicate* and *OperatorSaysSearch* stands for the human operator who accompanies the robot in the house search mission and is equivalent to the human operator issuing a specific obligation or prohibition. A prohibition is one of these obligations with a negation. For example, `OperatorSaysMove` is an obligation and ¬ `OperatorSaysMove` is a prohibition. The human operator is able to enable or disable certain policies in the Policy Engine. How this works is further defined in Chapter 4.3.

The complete policy list:

- DangerousImpliesCommunication **policy**: $D \supset \mathbf{O}comm$
  When there is an object with the status *dangerous* within sight of the robot, the robot is obligated to communicate its findings with the human operator.

- FireOrBomb **policy**: $(A \vee B) \wedge \neg C \supset \mathbf{O}move$
  When there is a fire or a bomb nearby, and no boobytrap, the robot is obligated to move outside of the building so it may reach a safe location.

- Boobytrap **policy**: $C \supset \mathbf{O}\neg move \wedge \mathbf{O}\neg search \wedge \mathbf{O}comm$
  When a boobytrap is nearby, the robot is obligated to communicate with the human operator and prohibited to move and prohibited to search (because moving or searching may trigger the boobytrap).

- ObligateMove **policy**: $(((operatorSaysMove) \vee A \vee B) \wedge \neg C) \supset \mathbf{O}move$
  This policy obligates the robot to move to a specific location in the virtual environment if the operator orders the robot to move (defined with $(operatorSaysMove)$), or there is a large fire or a bomb (specified with $A \vee B$), and there may be no boobytrap nearby (specified with $\neg C$). If these conditions hold true, the ObligateMove policy becomes active.

- ProhibitMove **policy**: $((\neg operatorSaysMove) \vee C) \wedge \neg(A \vee B) \supset \mathbf{O}\neg move$
  This policy prohibits the robot to move and stops all movement. The policy is activated when the operator orders the robot to stop moving or if there is a boobytrap nearby (denoted by $(\neg operatorSaysMove) \vee C$), and no large fire or a bomb nearby (denoted by $\neg(A \vee B)$).

- ObligateCommunicate **policy**: $((operatorSaysCommunicate) \vee D) \supset \mathbf{O}comm$
  This policy orders the robot to communicate with the human operator and is activated when the human operator wants to activate this policy with $(operatorSaysCommunicate)$, or if there is a dangerous object. Because a boobytrap is defined as a *dangerous object* in the ontology, it is not necessary to include the boobytrap individually.

- ProhibitCommunicate **policy**: $((\neg operatorSaysCommunicate) \wedge \neg D) \supset \mathbf{O}\neg comm$
  This policy prohibits the robot to communicate with the human operator and may be applied when a stressful situation requires the full attention of the human operator, but may only be

fired when there is no dangerous object nearby and the human operator consciously wants to activate this policy.

- `ObligateSearch` **policy**: $((operatorSaysSearch) \wedge \neg C) \supset \mathbf{O}search$
  This policy obligates the robot to search for weapons or other dangerous objects that are within sight of the camera of the robot, as long as the human operator wants to activate this policy and there is no boobytrap nearby.

- `ProhibitSearch` **policy**: $((\neg operatorSaysSearch) \vee C) \supset \mathbf{O}\neg search$
  This policy prohibits the robot to search for weapons or other dangerous objects, and may only be activated when the human operator wants to activate this policy or if there is a boobytrap nearby.

The `Boobytrap`, `FireOrBomb` and `DangerousImplicesCommuncation` policies will be (forcefully) activated by the robot if their conditions hold true, even if it means that it will overrule the opposite policy currently activated by the human operator. They will also prevent the activation of the opposing policy by the human operator.

4.1.4 Policies in Drools

Two opposing policies will be explained how they technically work; the `ObligateMove` policy and the `ProhibitMove` policy, followed by the conflict-resolving policies `DangerousImpliesCommuncation`, `FireOrBomb` and `Boobytrap`. A complete list of the policies in Drools code can be found in Appendix A.

```
rule "ObligateMove"
        when
                PolicyDecision(appliesToActor == $robot, appliesToAction ==
                    $action) or exists LargeFire() or exists Bomb(aboutToExplode
                    )
                not BoobyTrap()
                $robot : Robot()
                $action : MoveAction()
        then
                MoveObligationPolicyDecision obligation = new
                    MoveObligationPolicyDecision($robot, $action);
                insert(obligation);
        end
```

`ObligateMove` **policy**: $(((operatorSaysMove) \vee A \vee B) \wedge \neg C) \supset \mathbf{O}move$

This policy obligates the robot to move to a specific location in the virtual environment if the operator orders the robot to move, or there is a large fire or a bomb (specified with $A \vee B$), and there may be no boobytrap nearby (specified with $\neg C$). If these conditions hold true, the `ObligateMove` policy becomes active.

A policy is formulated in the way as described in Chapter 2.1: `when` certain conditions are met, `then` do some tasks. The Policy `ObligateMove` is interpreted as follows. The code `PolicyDecision(appliesToActor == $robot, appliesToAction == $action)or exists LargeFire()` `or exists Bomb(aboutToExplode)` is the Drools code for the deontic logic $(operatorSaysMove) \vee A \vee B$. With this, the policy checks if the human operator wants to activate this policy, or if there is a large fire or a bomb about to explode. The policy then makes sure that there is no boobytrap nearby, since moving the robot with a boobytrap nearby may trigger the boobytrap. This is done with the Drools code `not BoobyTrap()` and is the same as the deontic logic $\neg C$. The variable `$robot` is defined by the Java Class (or ontology class) `Robot()` and makes sure that this policy applies to the robot. The `$action` variable is linked to the Java Class `MoveAction()`. If these conditions are met (the policy applies to the robot and involves the action to move), then a new obligation is defined with `MoveObligationPolicyDecision obligation = new MoveObligationPolicyDecision($robot, $action);`, where the `$robot` variable is linked to the `Robot()` and the `$action` variable links to the class `MoveAction()` that makes

sure we are creating an obligation for movement. Finally, this new obligation for the robot with the move action is inserted in the active policy agenda with the code `insert(obligation)`.

```
rule "ProhibitMove"
when
        PolicyDecision(appliesToActor == $robot, appliesToAction ==
            $action) or exists BoobyTrap()
        not LargeFire() or not Bomb(aboutToExplode)
        $robot : Robot()
        $action : MoveAction()
    then
        MoveProhibitionPolicyDecision prohibition = new
            MoveProhibitionPolicyDecision($robot, $action);
        insert(prohibition);
    end
```

`ProhibitMove` **policy**: $((\neg operatorSaysMove) \vee C) \wedge \neg(A \vee B) \supset \mathbf{O}\neg move$

The policy `ProhibitMove` is roughly the same as the policy `ObligateMove`. The policy checks if the policy is activated by the human operator or if there is a boobytrap nearby. Then the robot and the action are defined. If these conditions are met, a new prohibition policy is created and inserted in the active agenda. In short, a *PolicyDecision* can be an obligation or a prohibition applied to a certain *Actor*, specifying a certain *Action*

```
rule "DangerousImpliesCommunication"
    when
        exists DangerousObject()
        $robot : Robot()
        $action : CommunicateAction()
    then
        CommunicateObligationPolicyDecision obligation = new
            CommunicateObligationPolicyDecision($robot, $action);
        insert(obligation);
    end
```

`DangerousImpliesCommunication` **policy**: $D \supset \mathbf{O}comm$. When there is an object with the status *dangerous* within sight of the robot, the robot is obligated to communicate its findings with the human operator.

The `DangerousImpliesCommunication` policy works the same as the `ObligateCommunicate` policy specified in Appendix A, the only difference found on line 4, `exists DangerousObject()`, that checks if a dangerous object exists. When the robot moves in the virtual environment and a dangerous object is within sight of the robot, this `exists DangerousObject()` becomes true and the conditions for this policy are met. If this is the case, a new obligation is created with `CommunicateObligationPolicyDecision obligation = new CommunicateObligationPolicyDecision($robot, $action);` that obligates the robot to communicate. The variable `$action : CommunicateAction()` is used here so the obligation knows the variable `$action` refers to the `CommunicateAction()`.

```
1  rule "FireOrBomb"
2          when
3                  exists LargeFire() or exists Bomb(aboutToExplode)
4                  not Boobytrap()
5                  $robot : Robot()
6                  $action : MoveAction()
7          then
8                  MoveObligationPolicyDecision obligation = new
                       MoveObligationPolicyDecision($robot, $action);
9                  insert(obligation);
10         end
```

`FireOrBomb` **policy**: $(A \vee B) \wedge \neg C \supset \mathbf{O}move$. When there is a fire or a bomb nearby, and no boobytrap, the robot is obligated to move outside of the building so it may reach a safe location by moving to safety.

The `FireOrBomb` policy checks if there is a fire or a bomb about to explode with the variables `exists LargeFire() or exists Bomb(aboutToExplode)` and makes sure there is no boobytrap with `not Boobytrap()`. The action for movement has been specified: `$action : MoveAction()`. This action is then used to create an obligation: a move obligation `MoveObligationPolicyDecision obligation = new MoveObligationPolicyDecision($robot, $action);` that uses `$action`, the action that entails the `MoveAction()`.

```
rule "Boobytrap"
        when
                exists Boobytrap()
                $robot : Robot()
                $action1 : MoveAction()
                $action2 : SearchAction()
                $action3 : CommunicateAction()
        then
                MoveProhibitionPolicyDecision prohibition = new
                    MoveProhibitionPolicyDecision($robot, $action1);
                insert(prohibition);
                SearchProhibitionPolicyDecision prohibition = new
                    SearchProhibitionPolicyDecision($robot, $action2);
                insert(prohibition);
                CommunicateObligationPolicyDecision obligation = new
                    CommunicateObligationPolicyDecision($robot, $action3);
                insert(obligation);
        end
```

`Boobytrap` **policy**: $C \supset \mathbf{O}\neg move \wedge \mathbf{O}\neg search \wedge \mathbf{O}comm$. When a boobytrap is nearby, the robot is obligated to communicate with the human operator and prohibited to move and prohibited to search (because moving or searching may trigger the boobytrap).

The `Boobytrap` policy checks if there is a boobytrap with `exists Boobytrap()`. Because the policy results in two prohibitions (no movement and no searching) and one obligation (obligated to communicate), three action variables are defined that can be used in the corresponding prohibitions and obligation policy decisions. `$action1 : MoveAction()`, corresponds with the `MoveProhibitionPolicyDecision` prohibition, `$action2 : SearchAction()` corresponds with the `SearchProhibitionPolicyDecision` prohibition and `$action3 : CommunicateAction()` corresponds with the `CommunicateObligationPolicyDecision` obligation.

The choice for these behaviors and their prohibition and obligation policies was made to illustrate this case study. The policies correspond with the behaviors in the behavior tree, and a policy corresponding to communication, movement or searching, will atleast occur once during a test run in the virtual environment. These basic behaviors are described in detail in Chapter 4.2.1.

4.2 Behavior Tree

Creating a behavior tree in Behavior Designer [15] in Unity gives us the flexibility of an intuitive visual editor. Basic behaviors can be created and linked together, combining these solitary behavior actions into adaptive, responsive behavior patterns. Examples of some basic behavior actions could be `Within Sight`, `Move Towards Location` and `Pick up Object`. Linking these behaviors together could create many different behavior patterns. For example, if a weapon is within sight, then move towards it and pick up the object. A different example could be: if a fire is within sight, then move towards the exit of the building.

4.2.1 Behavior Tree - Basic Behaviors

The following behaviors (actions and conditional behaviors) have been specified and used in our scenario. There are many more behaviors that can be implemented, but those were not deemed necessary for this scenario. The decision for these behaviors was a conscious decision to limit the complexity of the framework.

Move Towards Location. Moves towards a specific location, specified by a tag.

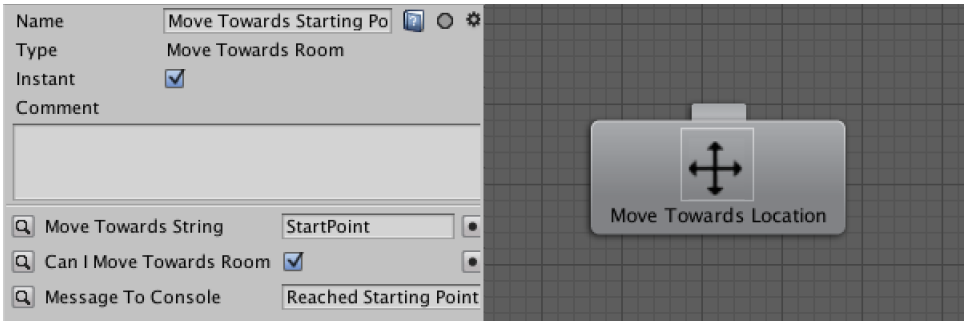


Figure 11: Properties of the Move Towards Location node

Figure 11 shows the properties of the `Move Towards Location` node on the left, and the node itself on the right. When a node is selected in the behavior tree, the properties of this particular node can be changed. The property screen is unique for each node and contains different property fields,

[15] `https://www.opsive.com/assets/BehaviorDesigner/`

depending on the type of node that is currently selected. The property field `Move Towards String` defines where to go and corresponds to an (in)visible object or location in the 3D environment with a certain tag. This tag is a string of text and can be used in this `Move Towards String` property field. A recognition point is placed outside the house with the tag `StartPoint`. The node returns `SUCCESS` once the robot is in close proximity to the location or object. The `Can I Move Towards Location` checkbox determines if this node can be fired at all, or not. The status of this checkbox is decided by the policy that is currently active; see Chapter 4.1. If the policy `ObligateMove` is active, this checkbox will be checked and the `Move Towards Starting Point` node will be able to fire. If the policy `ProhibitMove` is active, this node will not be active and automatically returns `FAILURE`. More about this policy influencing the behavior tree can be found in Chapter 4.3. The `Message to Console` property field sends the corresponding string to the console for logging and error reporting.

Human Instructions. Communicates a message to the human operator.

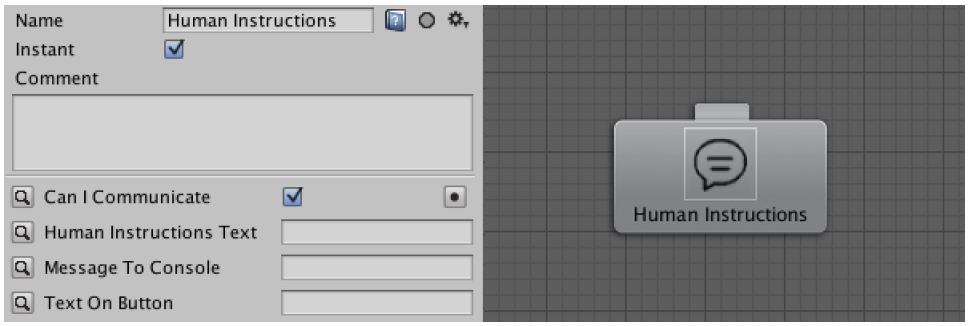


Figure 12: Human Instructions node - properties

Figure 12 shows the properties of the `Human Instructions` node. This is roughly the same as the `Move Towards Location` node. There is a checkbox corresponding to the currently active policy. If the policy `ObligateCommunicate` is active, the checkbox is checked, allowing this node to be run. The policy `ProhibitCommunicate` unchecks this checkbox, returning `FAILURE` upon running this node. The property field `Human Instructions Text` is the text that is displayed while playing the scenario. The property field `Text on Button` is a string that is printed on a clickable button. Once this button is clicked on the screen, the node returns `SUCCESS` and the behavior tree will continue with the next node in line. The `Message to Console` property field sends the corresponding string to the console for logging and error reporting.

Wait. Waits for a specific amount of seconds.

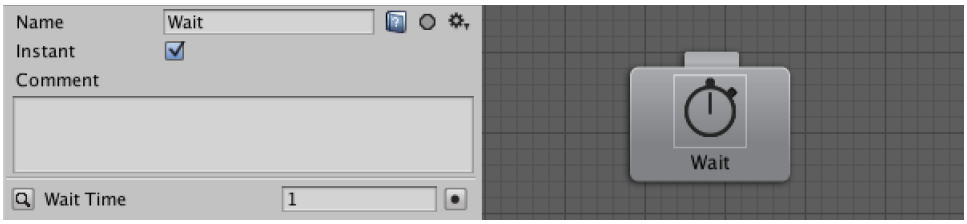


Figure 13: Wait node - properties

Figure 13 shows the properties of the `Wait` node. This node waits for a number of seconds, specified in the `Wait Time` property field. When the number of seconds has passed, the node returns `SUCCESS`.

Within Sight. This behavior checks if a certain object or item is within sight.

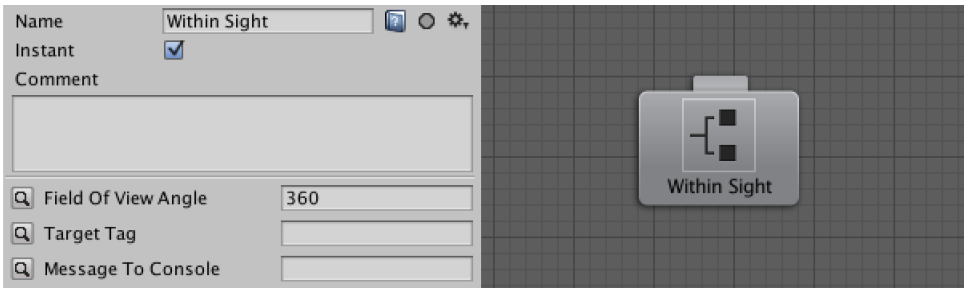


Figure 14: Within Sight node - properties

Figure 14 gives us an example of the `Within Sight` node. It scans the visible area around itself, comparable to how vision or a camera works. Once an object is within sight that has a tag that corresponds to the tag specified in the `Target Tag` property field, the node returns `SUCCESS`.

Open Door. Opens a door. This node is a behavior that only works when standing in close proximity to a closed door.

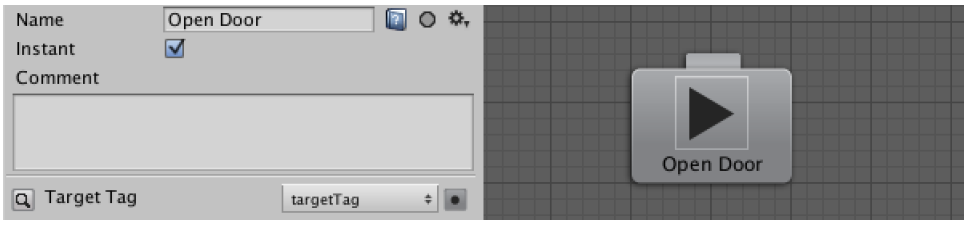


Figure 15: Open Door node - properties

Figure 15 shows the `Open Door` node. It has only one property field. The `Target Tag` property field uses a *shared variable* that is defined earlier in the behavior tree. The action to open a door is dependent on an earlier node that defines which door we are talking about, most likely a `Move Towards Location` node that moved towards a specific door. Then the `Open Door` node can open this specific door.

4.2.2 Behavior Tree - Sub-Trees Explained

The behavior tree will now be discussed, zooming in on different sub-trees within the complete tree.

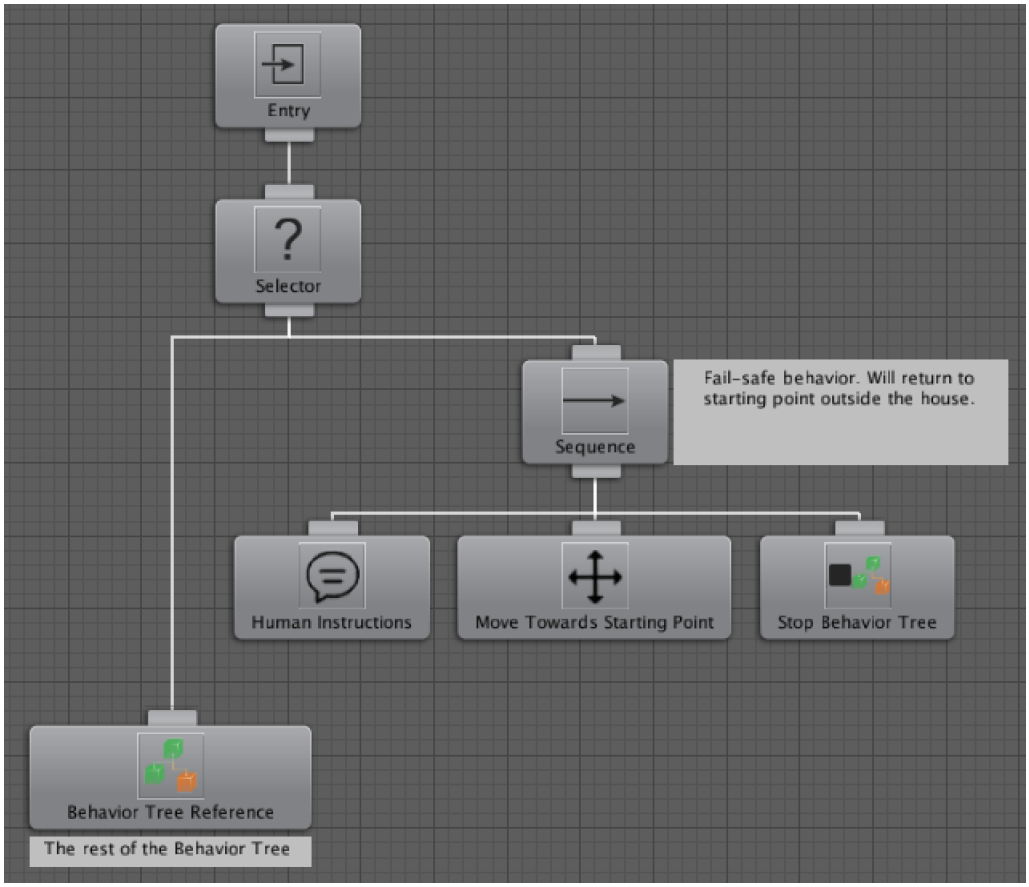


Figure 16: Behavior Tree - Entry

Figure 16 is the main tree that contains the rest of the sub-trees. The `Entry` node acts as the root of the tree and that is its only function. The `Behavior Tree Reference` is a node that contains another tree, minimizing the current visible tree for an optimized overview of the total tree (since the complete tree that is completely collapsed, and thus not minimized, can become very large). The `Selector` node is similar to an 'or' operator and will return `SUCCESS` as soon as one of its child nodes returns with `SUCCESS`. If a child node returns `FAILURE` then it will sequentially run the next task. If both child task returns `FAILURE` then the selector task will return `FAILURE`.

The `Sequence` node on the right of the selector task in Figure 16 is a fail-safe behavior that triggers when the complete behavior tree on the left fails. The sequence node will then resort to a single message 'Scenario aborted - Returning to starting point' (defined in the node

Human Instructions) that returns the robot to the starting point outside the house (defined in the node Move Towards Starting Point) and stops the behavior tree. This fail-safe behavior will only be triggered when the Behavior Tree Reference returns FAILURE ; this will normally (and ideally) not happen.

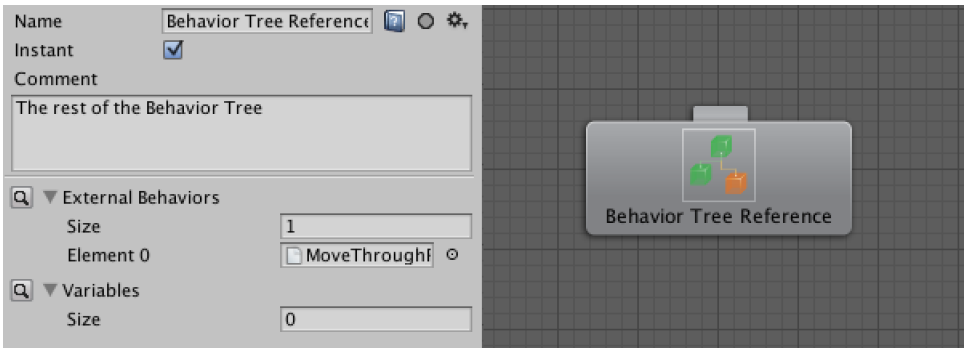


Figure 17: Behavior Tree Reference

The Behavior Tree Reference contains the rest of the behavior tree and the node is shown in Figure 17. The external behavior under Element 0 is called MoveThroughRooms . Variables can be defined by increasing the size and appointing a variety of different variables that are then used in the complete tree entailed by the behavior tree reference node. For example, a Bool variable (true or false) named CanIOpenTheDoor that is checked will keep this checkbox checked for every node in this behavior tree reference with the same variable name CanIOpenTheDoor . The behavior tree within this reference is too large to display here. See Appendix E for a visual representation. This sub-tree includes several other sub-trees, mainly behaviors that are created to express a specific behavior.

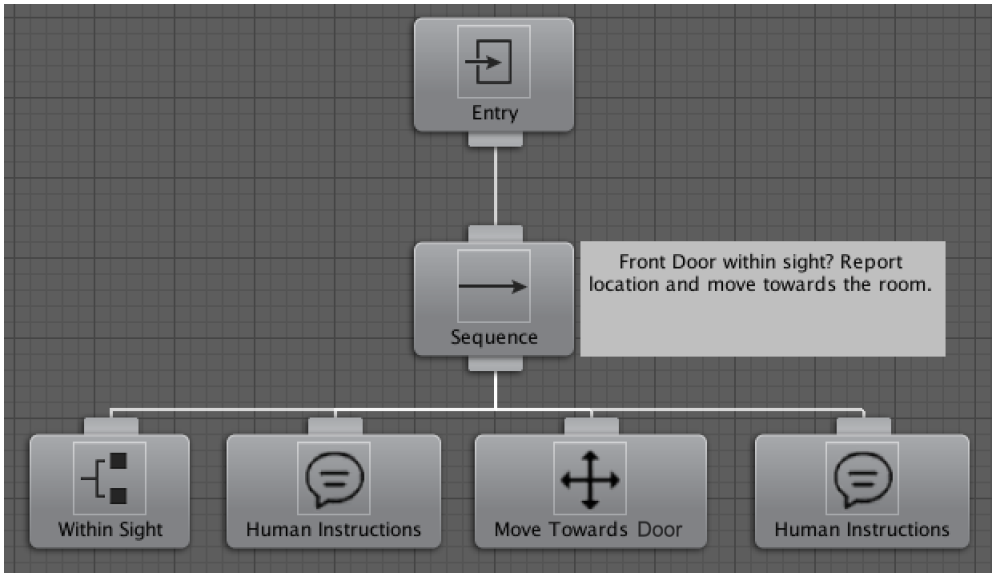


Figure 18: Sub-tree - front door behavior

The first in line is the sub-tree that defines the behavior that deals with the front door. Figure 18 is this sub-tree in a collapsed state. This sub-tree checks if the front door is within sight with the `Within Sight` node. If this node is triggered it returns `SUCCESS` and the `Human Instructions` node is triggered. The robot sends a message that it spotted the front door and after the human operator clicks the button to continue the mission, the node's status changes to `SUCCESS` and the robot will move towards the front door with the `Move Towards Door` node. The last `Human Instructions` node communicates that it reached the door and it will continue the mission.

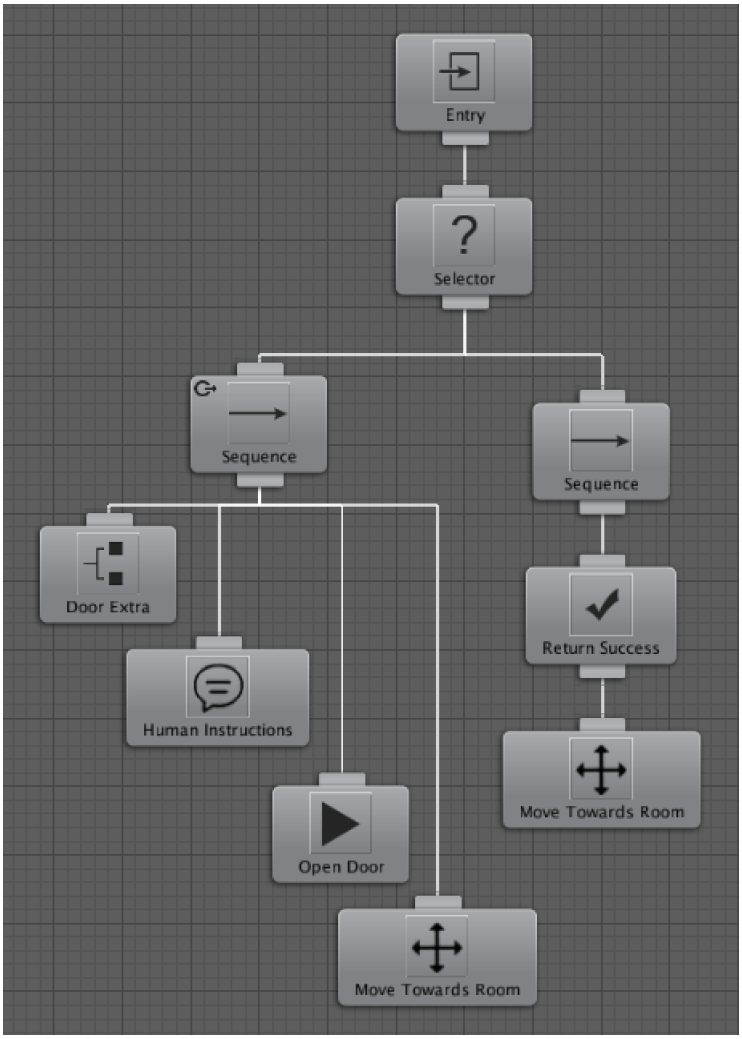


Figure 19: Sub-tree - moving through rooms

Figure 19 shows the sub-tree `MoveThroughRooms`. The right part of the selector node will be active first, in this case the `Move Towards Room` node. This sub-tree has a selector node where the left sequence is of a `lower priority` abort (defined by the icon in the top left corner in the `Sequence` node). Behavior trees can be organized from more important tasks to least important tasks. A lower priority abort will re-evaluate when any task to the right of the current branch is active. If a door is obstructing the movement towards the room, the right part of the selector node is aborted and the left part starts running. It will open a dialogue with the `Human Instructions` node,

communicating that a door is in its path and that it will open the door. After the human operator clicks the button to open the door, the `Open Door` action will open the door and the robot will eventually continue moving towards the room.

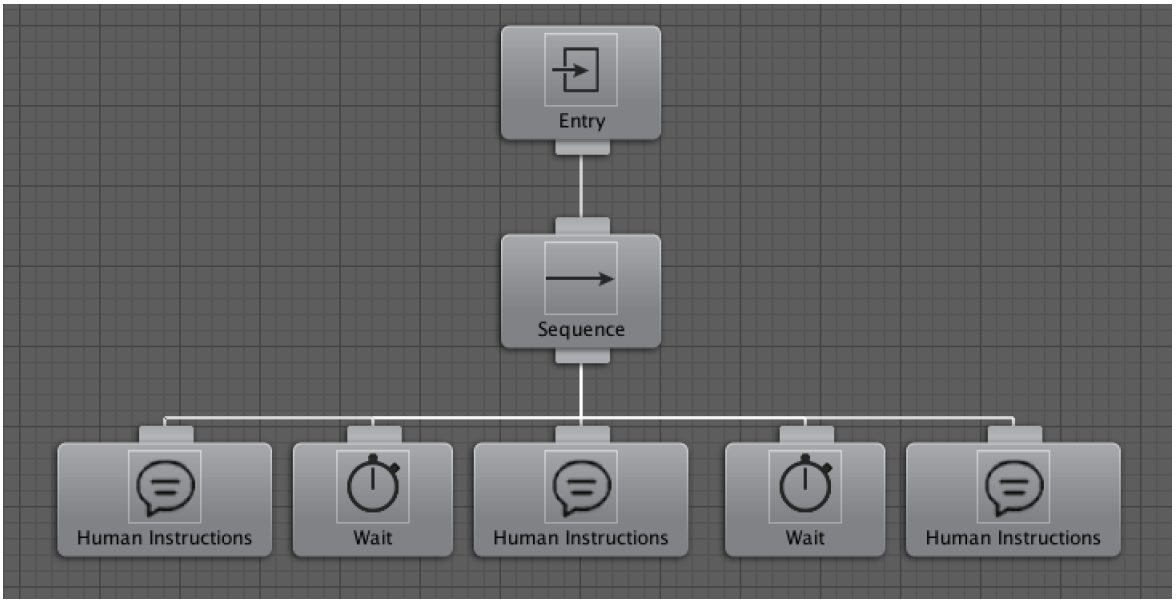


Figure 20: Sub-tree - Check the Stairs

Figure 20 contains the sub-tree that checks the stairs for boobytraps. It communicates with the user through the `Human Instructions` nodes before, during and after the scan. The `Wait` nodes will be running for 3 seconds each before returning `SUCCESS`.

4.3 Combining Policies and Behavior Trees

The combination of policies and behavior trees is done in several steps. As is shown in Chapter 4.2, sub-trees are captured in behavior tree references. A behavior tree reference node can hold a variety of variables, including those that communicate with the policies written in Drools. The same is applicable to single nodes. See figure 11 for the `Move Towards Location` node or figure 12 for the `Human Instructions` node. These nodes work with a bool that if checked, the `Move Towards Location` node or `Human Instructions` node are allowed to be executed. If unchecked, they will automatically return `FAILURE`. The true-or-false states are influenced by the Drools policies. When the policy

ObligateMove is active, the bool in the behavior tree will be true and the corresponding nodes are allowed to have the status RUNNING or SUCCESS. Once the policy ProhibitMove is active, the bool in the behavior tree will return false and the corresponding node or sub-tree will return FAILURE, continuing with the next node or sub-tree in line.

In collaboration with TNO, the *Policy Engine TNO* was developed, or PET in short. The main function of PET is to infer decisions for all connected agents based on available ontology classes. An ontology defines the relations between objects by inheritance. For example, a Robot is a child of an Agent, or a Bomb is a child of a Dangerous object and can have the status About to explode. Every object used is an instance of an ontology class, creating a clear structure of hierarchy and relations for both the user and the system to understand how data and objects are related to each other. Policies are not limited to a specific ontology, but do require a basic ontology. These are the ontology classes Action and Actor, allowing policies to always address an agent as an actor performing a certain action.

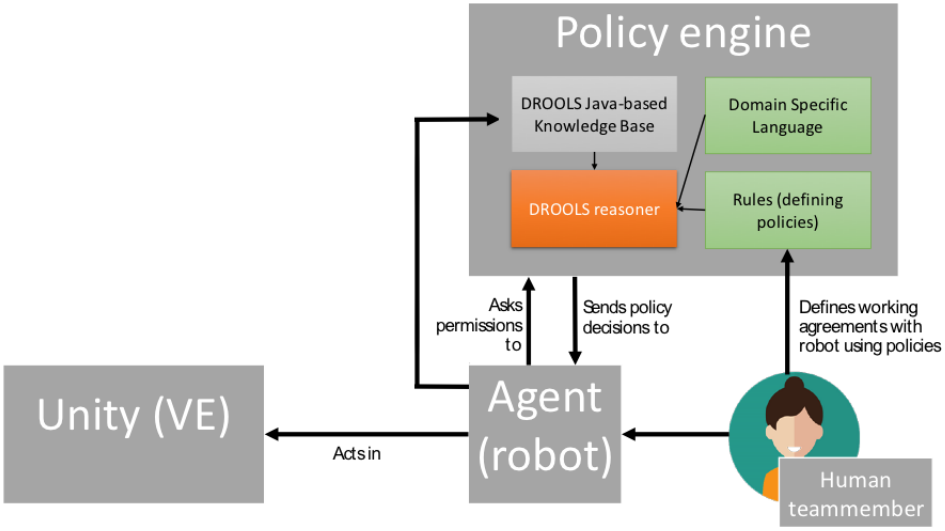


Figure 21: Policy Engine HRT

The complete model can be visualized as shown in figure 21. The agent (the robot) asks for policy decisions to the policy engine. The human interacts with the robot, and defines policies using the policy engine. The agent is connected to Unity, the virtual environment in which it acts. The agent

shares its knowledge in the Drools knowledge base. Objects in the virtual environment are classified, corresponding with ontology classes in the ontology knowledge base. This way, the robot may encounter a weapon and knows it is a weapon, and what its relation is compared to other objects. A boobytrap can be recognized in the virtual environment, triggering the `Boobytrap` policy that overrules any other policies previously activated.

PET comes with a user interface, PETUI in short. The PETUI is a Java application written in Eclipse [16]. This policy engine hosts the policies written in Drools and is capable of communicating these policies with Unity through a websocket connection. An asset from the Unity Asset Store called Socket.IO [17] was used to establish this connection. Websockets provide a persistent connection between a client and server that both parties can use to start sending data at any time. They provide a persistent, low latency connection that can support transactions initiated by either the client or server.

A brief description of the PETUI will follow. These can also be found in Appendix D, showing them with annotations and without annotations.

[16] `http://www.eclipse.org/`
[17] `https://www.assetstore.unity3d.com/en/#!/content/21721`

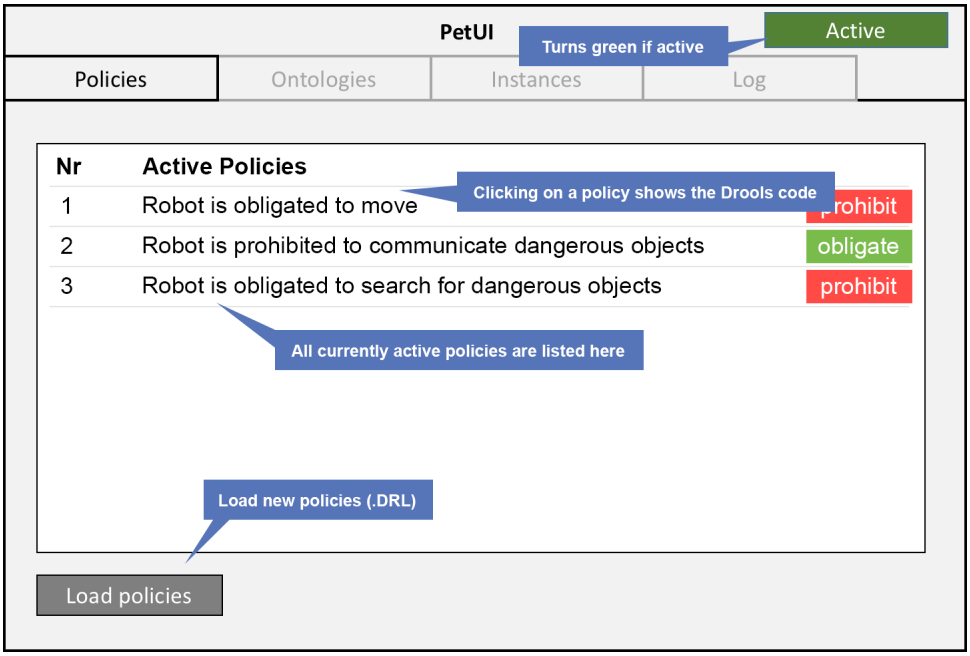


Figure 22: PETUI - overview of the active policies.

Figure 22 shows an overview of the PETUI with the policies that are currently active. Clicking a policy will show the Drools rule in code. New policies can be loaded into the policy engine, creating different strategies for a different approach. The specific obligation or prohibition policy that is currently active can also be overruled by its opposing policy by clicking the green or red button at the end of each line, switching the approach from an obligation towards a prohibition or the other way around. This way, the human operator can change the appropriate strategy at real-time by controlling basic behaviors for the robot. As discussed, the `DangerousImpliesCommunication`, `Boobytrap` and `FireOrBomb` policies are activated when certain situations arise during the house search. When this happens, the robot will overrule the previously activated policies by the human operator and makes it temporarily impossible for the human operator to change these policies. How they are activated and how they correspond with the behavior tree is discussed in Chapter 5.

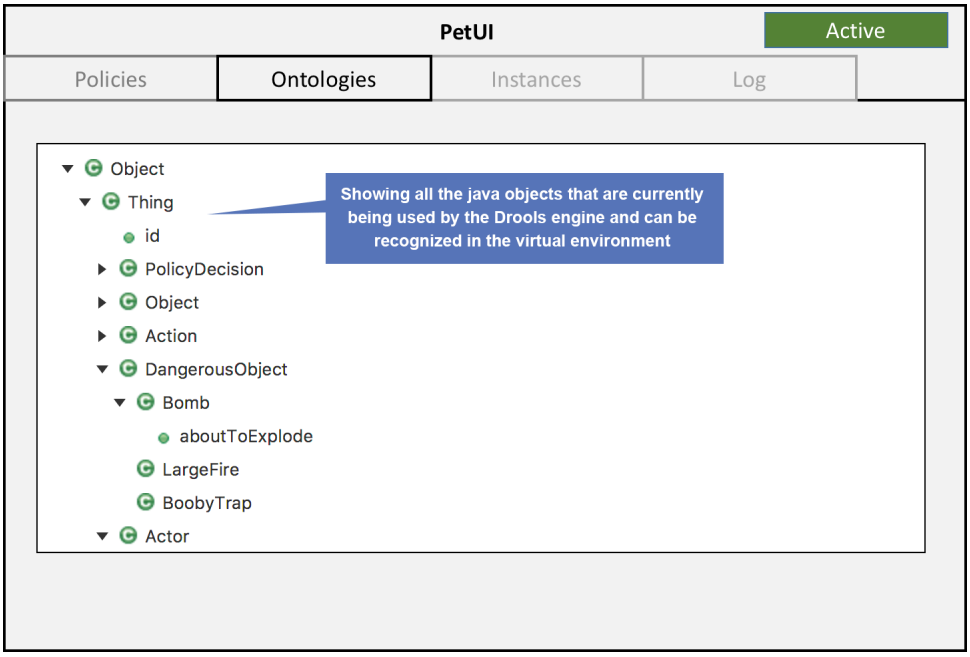


Figure 23: PETUI - overview of the active ontology classes.

Figure 23 shows the PETUI with the ontology classes that are used in the scenario and that are active. A full list of the ontology classes in a tree view used can be found in Appendix C. Objects in the virtual environment are classified with tags, so that when the robot encounters a bomb, it knows this is classified as a dangerous object. With ontology classes, the robot can understand its environment and may overrule policies issued by the human operator if it deems this is beneficial.

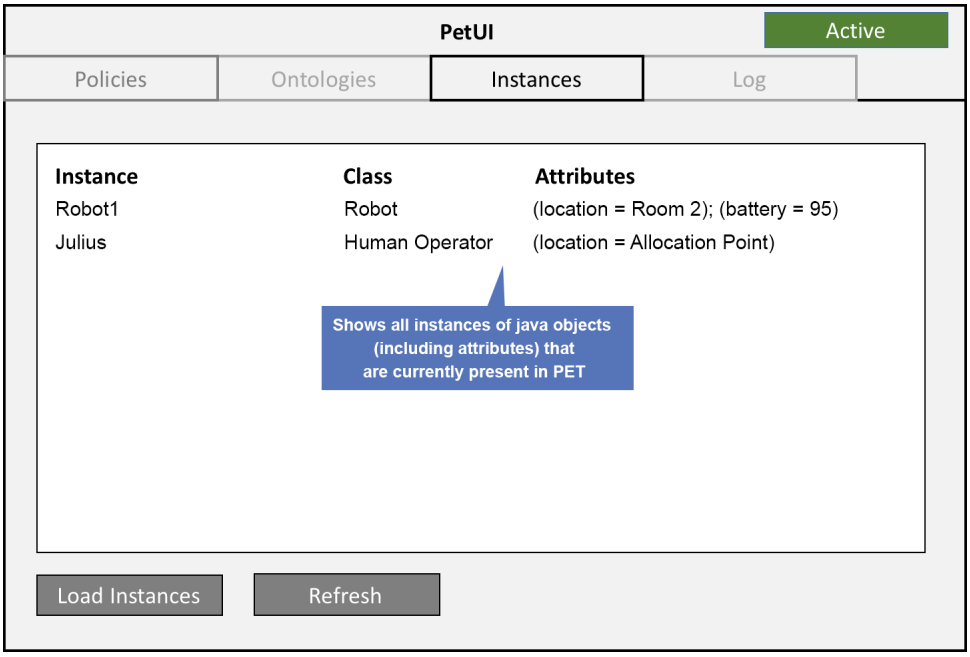


Figure 24: PETUI - overview of the active instances.

Figure 24 shows the PETUI with the active instances. Currently, there are not many instances included in the virtual environment. These instances may vary, depending on the situation, and they hold some knowledge about variables that can change during the house search mission. Here, we illustrated two instances, communicating the location of the human operator and the robot, and the battery percentage of the robot. When using multiple robots, they will also be listed here.

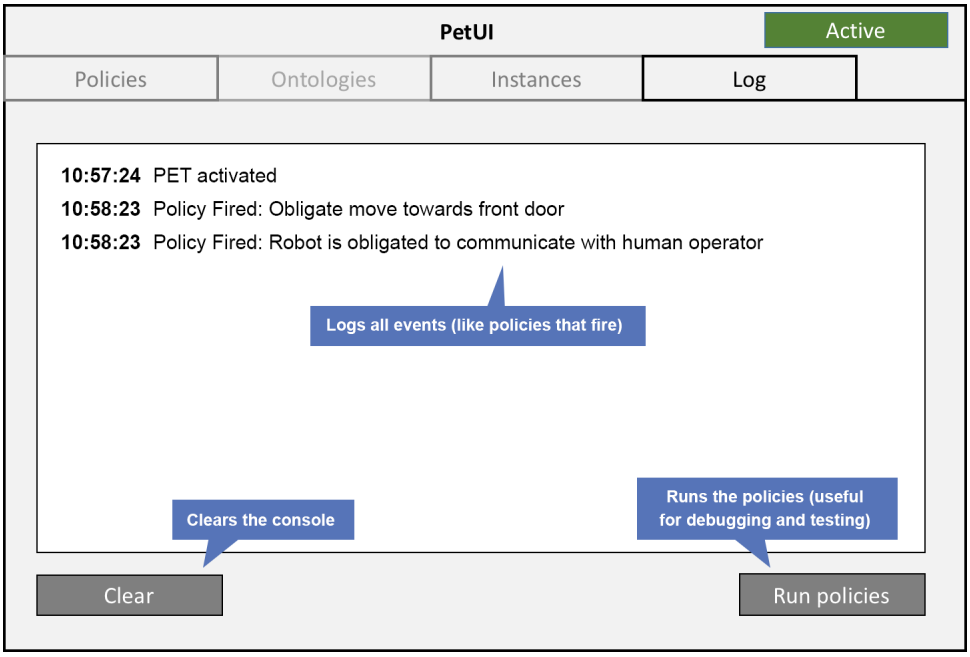


Figure 25: PETUI - overview of the log.

Figure 25 represents the log of the PETUI. The log shows an event that has occurred at a certain point of time. The log can be refreshed with the use of a button and will automatically be reset when the policy engine is closed.

5 Simulation Run

In this chapter, a detailed description is given on each step that the human operator will take and how the scenario develops through time. First, the policy engine will have to be launched. This is done in a stand-alone Java application where the websocket client is started that allows the communication between Unity and the policies written in Drools.

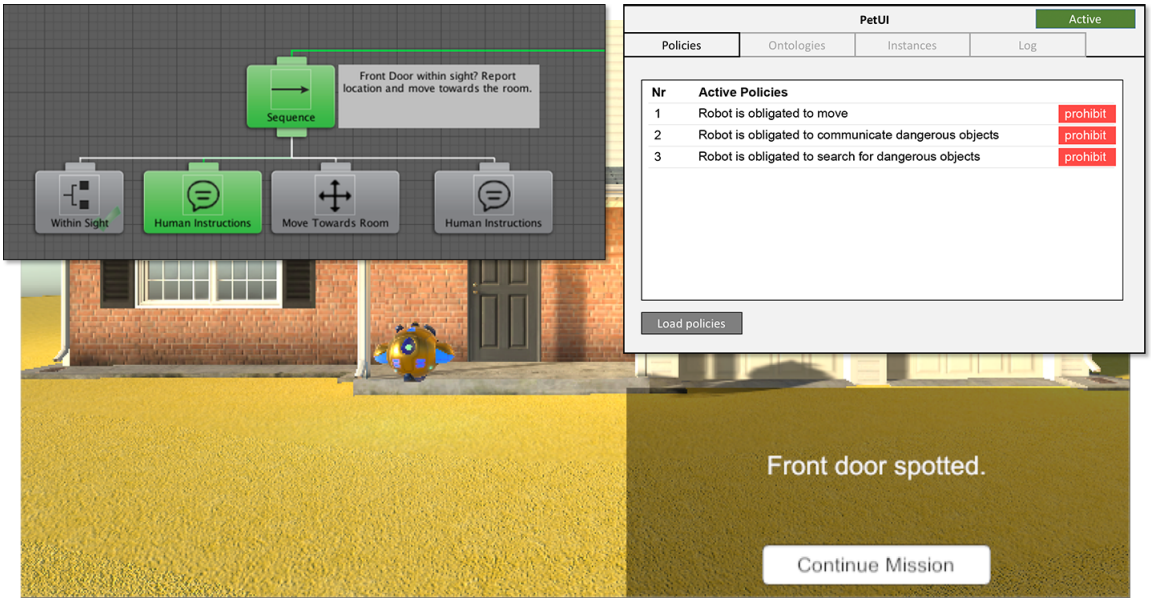


Figure 26: Simulation run - front door behavior.

When the policy engine is active, the virtual world in Unity can be activated. This can be done in Unity or as a stand-alone application. When both the virtual environment in Unity and the policy engine are active, the simulation run is started. Figure 26 shows the beginning of the simulation run. The policy engine is shown as a stand-alone application in the top-right corner and the sub tree of the behavior tree that corresponds with the behavior happening on-screen is shown in the top-left corner. The first action of the robot is to check if the front door is within sight. If this is true, the robot will issue a communication screen with the human operator (the current active node in Figure 26), followed by moving towards the front door. When the robot is at the front door, the human operator is asked to continue the mission when ready.

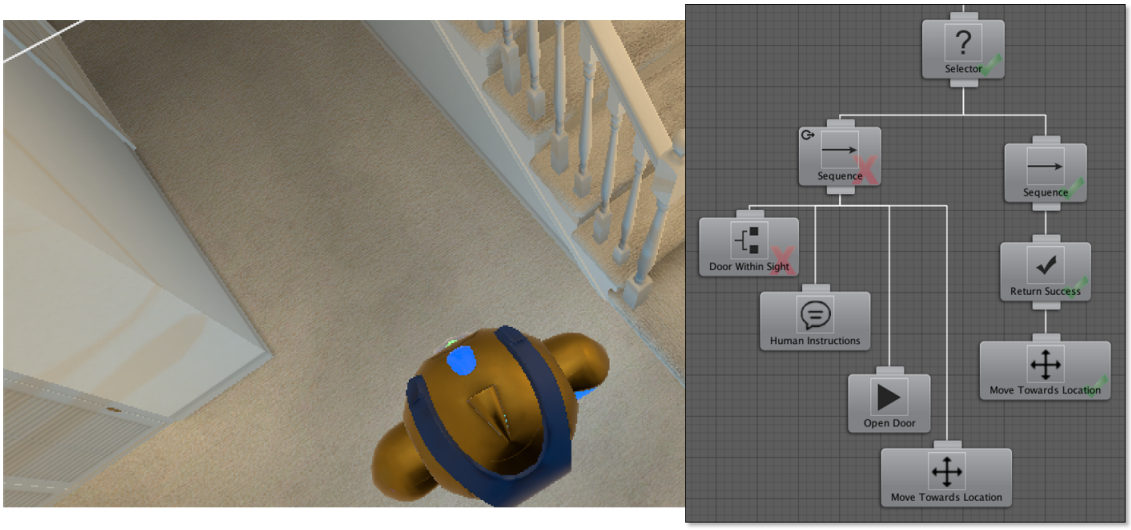


Figure 27: Simulation run - movement to hallway.

Figure 27 shows the movement from the front door to the hallway. Because there is no door obstructing the possibility to reach this hallway location, the right side of the behavior tree is executed. See Figure 19 in Chapter 4.2.2 for a reminder how this particular sub-tree works.

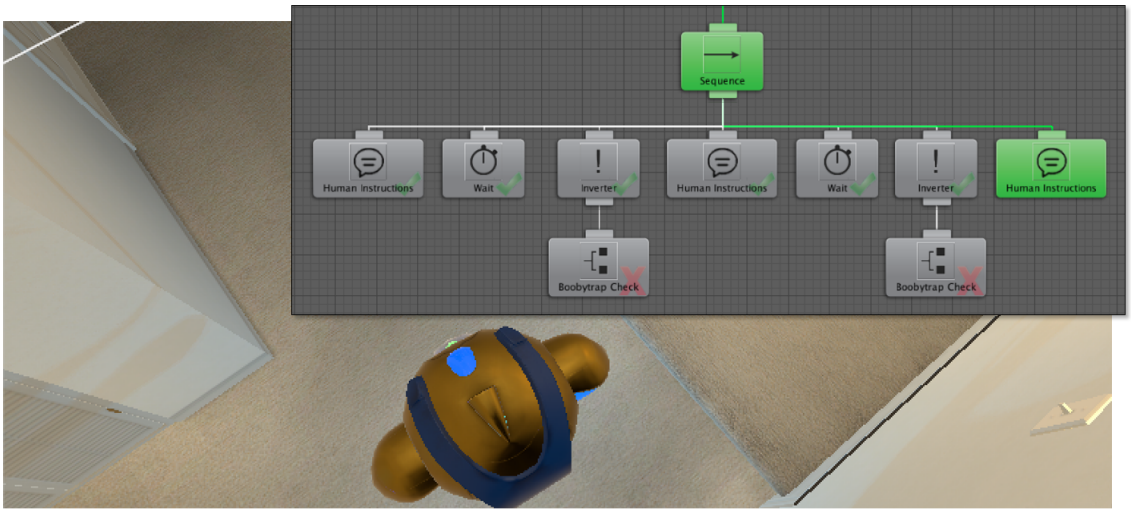


Figure 28: Simulation run - boobytrap check.

Figure 28 is executed after reaching the hallway. The robot performs two boobytrap checks: one

for the stairs going up and one for the route towards the second door. The `inverter` node makes sure that if there are no booytraps found, the node returns `SUCCESS` so the tree can fire the next node in line. If there is a boobytrap, this particular sub-tree returns `FAILURE` and the boobytrap policy will be activated.

Figure 29: Simulation run - consolidation point and search order.

Figure 29 shows the situation in the first room. The robot creates a consolidation point and is visualized in the virtual environment with a large floating arrow. Then the order of searching the room is defined: the robot scans the left area of the room, the human operator will search the right side of the room.

The robot will continue to walk through the house, moving from room to room as shown in Chapter 3.2. Instead of discussing each single room (which would be a repetitive process), the following scenarios describe particular situations that may occur during the simulation run in different rooms. These situations entail the possibility of encountering a boobytrap (triggering the `Boobytrap` policy), a large fire or a bomb about to explode (triggering the policy `FireOrBomb` policy) and a weapon (classified as a dangerous object, triggering the `DangerousImpliesCommunication` policy).

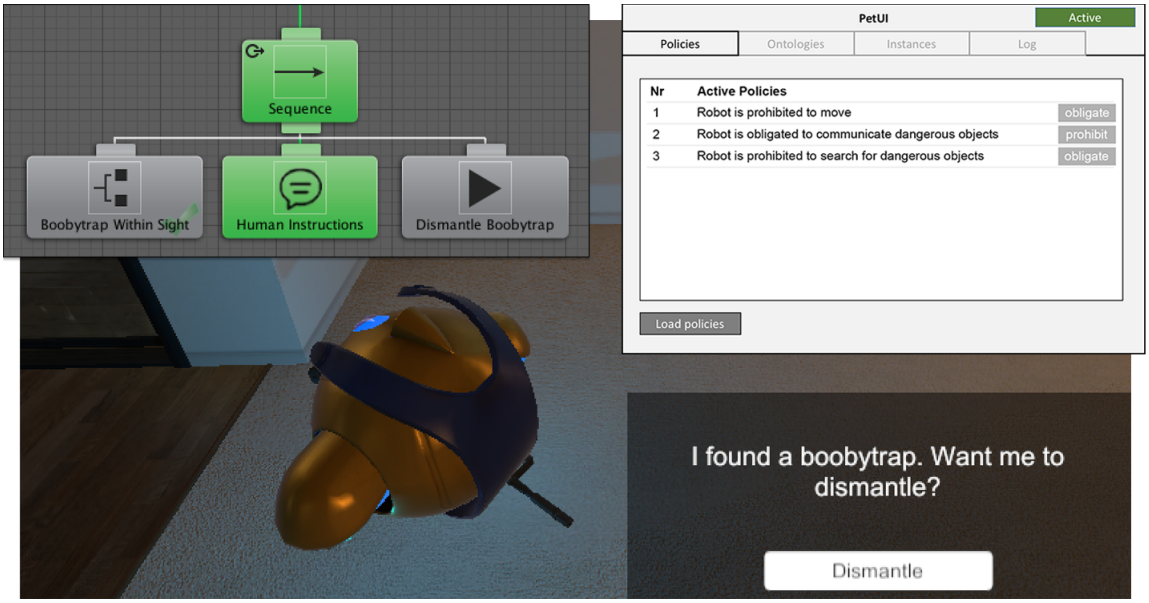


Figure 30: Simulation run - boobytrap.

Figure 30 shows the activation of policies in the policy engine and the corresponding behavior tree when a possible boobytrap has been spotted. The robot is able to recognize a boobytrap because the object in the virtual environment has a tag *boobytrap* and the ontology tells the robot how to classify and interpret a boobytrap in relation to other objects. See Appendix C for an overview of the ontology. When a boobytrap is spotted, a specific sub-tree becomes active that handles boobytraps. As discussed in Chapter 4.1.2, when the robot encounters a boobytrap, the `Boobytrap` policy enforces the robot to adhere to a specific set of policies: the robot is obligated to communicate, and prohibited to move and prohibited to continue searching. This is visualized in the policy engine as can be seen in Figure 30. The buttons to change a policy are turned grey to indicate that the human operator is not allowed to change these policies until the boobytrap sub-tree has has been completed.

Figure 31: Simulation run - dangerous object.

Figure 31 shows the behavior tree for a dangerous object. The robot starts a sequence when a dangerous object is within sight, in this case a weapon. The robot moves towards the specific object and communicates its findings with the human operator. Due to the ontology specified in the policy engine, the robot knows a weapon is classified as a dangerous object. See Appendix C for the complete ontology. In this case, the policy `DangerousImpliesCommunication` becomes active, where the robot overrules any policies previously specified by the human operator and forcefully activates the `ObligateCommunicate` policy. See Figure 31 for the activation in the policy engine for this `ObligateCommunicate` policy that momentarily cannot be turned off. The human operator can issue the command to pick up the weapon. After this, the robot continues the search and the human operator is now able to overrule the `ObligateCommunicate` policy with the `ProhibitCommunicate` policy if he or she deems this necessary.

Figure 32: Simulation run - large fire.

Figure 32 shows the behavior tree for a bomb about to explode or a large fire, as long as there is no boobytrap within sight. This corresponds with the `FireOrBomb` policy, see Chapter 4.1.4 for a reminder for more information. In this particular case, a bomb that is about to explode has been spotted. The current behavior tree checks if a large fire or a bomb about to explode is within sight. If so, it moves closely enough to examine the situation. It will then open a communication screen with the human operator, stating it will move outside of the building to avoid damage while also advising the human operator to do the same. Once outside, it will ask the human operator if he or she wants to proceed with the mission. As can be seen in the policy engine in Figure 32, the policies `ObligateMove` and `ObligateCommunicate` are active and cannot be overruled by the human operator.

6 Conclusion & Discussion

The research questions formulated in Chapter 1.4 were:

1. How can a framework for robots be created that simplifies the off-line creation and modification of adaptive behavior?
   (a) How can this technically be realized?
   (b) How can this be operationally embedded?

The adaptive behavior was achieved by combining policies and behavior trees, where certain policies based on an ontology allowed the robot to overrule any policies previously issued by the human operator. The technical realization of this framework has been done by writing policies in Drools and creating a behavior tree in Unity, with sub-trees representing specific behaviors in a modular fashion. The combination of these two was made possible by the Policy Engine TNO (PET), connecting the policies and the behavior trees through a websocket connection. In a small scenario we were able to create a collaborative task between a human operator and a robot, where a virtual environment allowed them to collaboratively perform a house search mission. This scenario could be implemented with the use of a policy engine, with policies written in Drools, and a behavior tree implementation in Unity. The policy engine and the behavior tree made it possible to implement adaptive behavior without losing explainability, where the robot followed the strategy chosen by the human operator, with some exceptions where the robot overruled the human operator when it deemed these new strategies to be more important. These exceptions have been written as policies in Drools and functioned well when used simultaneously with the other policies. The complete house search and the triggering of these specific policies has been proven to work in Chapter 5, where a simulation run has been discussed step by step. The case study discussed in this thesis shows that it is possible to combine policies and behavior trees in a scenario, creating a framework for the creation of adaptive behavior. Because the framework was developed in close collaboration with domain experts working with TNO, it has been found that the framework is operationally embedded and could hold value in future missions.

The complexity of the behavioral model of the agent correlates with the complexity of the environment. A more complex environment has more variables influencing the effectiveness of the

strategy chosen by a robot. It was a conscious choice to restrict the possible behaviors and policies to the ones discussed in this thesis, to research the value of a framework that combines behavior trees and policies. A more complex, uncontrollable environment is believed to lead to requiring more behaviors and policies (in quantity and quality) in order for the agent to behave successfully.

To summarize, three policies and their counterparts were chosen to illustrate this scenario. Each *obligation* policy has an opposite *prohibition* policy. The policies dictate about (1) moving, (2) communicating and (3) searching. It is not possible for a robot to apply to both the `ObligateMove` and the `ProhibitMove` policy, since one dictates the robot should move, while the other dictates the robot is not allowed to move. Several policies were introduced that function as a conflict-solving mechanism, giving the robot the possibility to autonomously overrule any policies issued by the human operator in certain specific situations. It is believed that following these strategies is crucial for the success rate of the mission and the robot will always adhere to these specific rules. By using an ontology, the robot is able to classify and categorize objects in the virtual environment in relation to other objects. The use of an ontology gives the robot a sense of understanding the current situation, choosing its own strategy over that of the human operator if it deems this necessary. This way, the human operator may activate the policy `ObligateMove`, while the robot may overrule this policy with `ProhibitMove` if it believes this policy to be crucial at that moment. Our case gives an example that this may happen if a boobytrap is nearby, since movement may trigger the trap. The `Boobytrap` policy also makes sure the robot stops searching for any other objects and obligates the robot to communicate the finding of the boobytrap with the human operator.

Discussion

There is no consensus yet amongst researchers on what is the best way to create adaptive robotic behavior. Current research of robotic behavior focusses on behavior-based robotics with a form of machine learning, where the robot learns how to behave with data in a given scenario. This can be done by trial and error, such as with reinforcement learning, or rule-based machine learning, where the robot identifies, learns or evolves rules to store, manipulate or apply knowledge (Witten, Frank, Hall, & Pal, 2016). There are many applications of machine learning to facilitate adaptive robotic behavior. Our approach is not focused on the learning capability of the robot, but more on the ease of implementation of adaptive behavior for the robot. A robot in a militaristic situation is not allowed to learn by trial and error, since an error could lead to severe injuries or even death. Behavior strategies should be pre-defined, where the robot is able to do what the human operator wants it to do, but overrule their authority in certain situations. It should always have a certain set of behaviors that allow it to behave autonomously to a certain extent. Ideally, the realization of this framework should be a workable example that can be used in future militaristic scenarios. The virtual test environment has been created in collaboration with TNO and other domain experts and resembles a real world scenario that can be used to gain insights in implications of these robots on current and future operating procedures. It will also enable the Defence organization to experience a wider range of future operational robot scenarios, expanding potential beneficial use of robots in the future. The combination of policies in Drools and behavior trees appears to be a novel concept and could encourage the community to further explore the possibilities and boundaries of this policy-driven behavior tree combination.

The *Policy Engine TNO (PET)* has been developed by TNO and its development was not part of this master thesis. The PET and the PETUI have not been developed to its full potential. The PETUI as discussed in Chapter 4.3 and shown in Appendix D functions as a guideline to how the PETUI should work in its final state. The PET was limited in its application power while this thesis was written during the research period at TNO. Because of time constraints and implementation difficulties, the deontic logic rules described in detail in Chapter 4.1.2 were implemented in PET, but were developed *after* the research period at TNO. The implementation of the ontology and these extra policies was done without the guidance of TNO. Because of my limited knowledge on Java (the PET was written in Java), extra time was needed to become familiar with the policy engine

and the connection with the behavior tree in Unity. The implementation of some conflict-resolving policies and the usage of the ontology was deemed necessary for this research and in the final state, it was possible to implement these successfully. These extra policies illustrate the possible strategies the robot could use to handle conflicting policies, by overruling the human operator.

Policies generally require application-specific information to reason over, forcing researchers to create policy languages that are bound to the domains for which they were developed. This prevents policy languages from being flexible or being applicable across domains. In order to enable agents to function well in dynamic and pervasive environments, which consist of different domains and systems and to understand and interpret policies correctly, we propose that they are presented in a semantic language like OWL [18]. Currently, the ontology-data is stored in the Drools knowledge base and each ontology class is written in Java. A semantic language like OWL allows different systems to share a model of policies, roles and other attributes, while the current ontology base is specifically designed for the purpose of this research. Future research may also focus on expanding the ontology knowledge base, creating a more diverse scenario that allows for more realistic adaptive behavior.

It is much easier to write and read rules than to write and read code. Future research may focus on applying Domain Specific Languages (DSLs). By creating Domain Specific Languages that model the problem domain, it is possible to write rules that are very similar and closely correlated with natural language [19]. By creating a DSL file for our Drools rule database, it is possible to transform Drools constructs to DSL sentences. Currently, there is no usage of a DSL due to implementation difficulties. Integrating a DSL into the Drools rule base will make it easier to read and write rules and could prove to be beneficial for the simplification of creating and changing current policies.

In the real world, a house search is often executed in a squad with multiple team members. This case study only uses one robot and one human operator. To create a framework that operates in a more realistic scenario, it is recommended to simulate a house search with more than one robot and/or human team member. This has currently not been investigated, because the aim was to

[18] https://www.w3.org/OWL/

[19] https://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch05.html#d0e6217

create a workable example for the combination of policies and behavior trees. The choice for one robot and one human operator was also to limit the complexity of the model. The usage of multiple robots will require a different policy set and the active policy set may be different per robot. The current policy engine is not capable of processing these kind of situations. Furthermore, it will be difficult to implement this framework in a real world scenario. Currently, the recognition of objects by the robot is simply done when an object with a specific tag (such as *weapon*) is within sight of the robot. The robot will always recognize every object within sight as the correct object. In a real world application, this will prove to be much more difficult since there is much more ambiguity in interpreting situations correctly. A boobytrap is supposed to be hidden until triggered, and the application of computer vision will prove to be difficult to recognize a boobytrap correctly in these situations. Looking back, it might be more realistic to create a set of policies that adhere to more easily-measured objects or situations. For example, the measurement of temperatures through certain sensors may lead to the conclusion that a fire is spreading, instead of recognizing objects through vision.

Appendix

A Drools Policies - Code

```
rule "DangerousImpliesCommunication"
        when
                exists DangerousObject()
                $robot : Robot()
                $action : CommunicateAction()
        then
                CommunicateObligationPolicyDecision obligation = new
                    CommunicateObligationPolicyDecision($robot, $action);
                insert(obligation);
        end
```

```
rule "FireOrBomb"
        when
                exists LargeFire() or exists Bomb(aboutToExplode)
                not Boobytrap()
                $robot : Robot()
                $action : MoveAction()
        then
                MoveObligationPolicyDecision obligation = new
                    MoveObligationPolicyDecision($robot, $action);
                insert(obligation);
        end
```

```
rule "Boobytrap"
        when
                exists BoobyTrap()
                $robot : Robot()
                $action1 : MoveAction()
                $action2 : SearchAction()
                $action3 : CommunicateAction()
        then
                MoveProhibitionPolicyDecision prohibition = new
                    MoveProhibitionPolicyDecision($robot, $action1);
                insert(prohibition);
                SearchProhibitionPolicyDecision prohibition = new
                    SearchProhibitionPolicyDecision($robot, $action2);
                insert(prohibition);
                CommunicateObligationPolicyDecision obligation = new
                    CommunicateObligationPolicyDecision($robot, $action3);
                insert(obligation);
        end
```

```
rule "ObligateMove"
        when
                PolicyDecision(appliesToActor == $robot, appliesToAction ==
                    $action) or exists LargeFire() or exists Bomb(aboutToExplode
                    )
                not BoobyTrap()
                $robot : Robot()
                $action : MoveAction()
        then
                MoveObligationPolicyDecision obligation = new
                    MoveObligationPolicyDecision($robot, $action);
                insert(obligation);
        end
```

```
rule "ProhibitMove"
        when
                PolicyDecision(appliesToActor == $robot, appliesToAction ==
                    $action) or exists BoobyTrap()
                not LargeFire() or not Bomb(aboutToExplode)
                $robot : Robot()
                $action : MoveAction()
        then
                MoveProhibitionPolicyDecision prohibition = new
                    MoveProhibitionPolicyDecision($robot, $action);
                insert(prohibition);
        end
```

```
rule "ObligateCommunicate"
        when
                PolicyDecision(appliesToActor == $robot, appliesToAction ==
                    $action) or exists DangerousObject()
                $robot : Robot()
                $action : CommunicateAction()
        then
                CommunicateObligationPolicyDecision obligation = new
                    CommunicateObligationPolicyDecision($robot, $action);
                insert(obligation);
        end
```

```
rule "ProhibitCommunicate"
        when
                PolicyDecision(appliesToActor == $robot, appliesToAction ==
                    $action) and not DangerousObject()
                $robot : Robot()
                $action : CommunicateAction()
        then
                CommunicateProhibitionPolicyDecision prohibition = new
                    CommunicateProhibitionPolicyDecision($robot, $action);
                insert(prohibition);
        end
```

```
rule "ObligateSearch"
        when
                PolicyDecision(appliesToActor == $robot, appliesToAction ==
                    $action) and not BoobyTrap()
                $robot : Robot()
                $action : SearchAction()
        then
                SearchObligationPolicyDecision obligation = new
                    SearchObligationPolicyDecision($robot, $action);
                insert(obligation);
        end
```

```
rule "ProhibitSearch"
        when
                PolicyDecision(appliesToActor == $robot, appliesToAction ==
                    $action) or exists BoobyTrap()
                $robot : Robot()
                $action : SearchAction()
        then
                SearchProhibitionPolicyDecision prohibition = new
                    SearchProhibitionPolicyDecision($robot, $action);
                insert(prohibition);
        end
```

B Unity - C# scripts

B.1 WithinSight.cs

```
1  using UnityEngine;
2  using BehaviorDesigner.Runtime;
3  using BehaviorDesigner.Runtime.Tasks;
4
5  public class WithinSight : Conditional
6  {
7          // How wide of an angle the object can see
8          public float fieldOfViewAngle;
9
10         // The tag of the targets
11         public string targetTag;
12
13         // Set the target variable when a target has been found so the
               subsequent tasks know which object is the target
14         public Transform target;
15
16         public string messageToConsole;
17
18         // A cache of all of the possible targets
19         private Transform[] possibleTargets;
20
21         int testval = 0;
22
23         public override void OnAwake()
24         {
25                 // Cache all of the transforms that have a tag of targetTag
26                 var targets = GameObject.FindGameObjectsWithTag(targetTag);
27                 possibleTargets = new Transform[targets.Length];
28                 for (int i = 0; i < targets.Length; i++) {
29                         possibleTargets[i] = targets[i].transform;
30                 }
31         }
32
33         public override TaskStatus OnUpdate ()
34         {
```

```
35              // Return success if a target is within sight
36              while (testval < possibleTargets.Length) {
37                      if (withinSight(possibleTargets[testval],
                            fieldOfViewAngle)) {
38                              // Set the target so other tasks will know which
                                    transform is within sight
39                              target = possibleTargets[testval];
40                              Debug.Log (messageToConsole);
41                              return TaskStatus.Success;
42                      }
43              }
44              return TaskStatus.Failure;
45      }
46
47
48      public void pickNext ()
49      {
50              if (testval < possibleTargets.Length)
51              {
52                      testval++;
53              }
54      }
55
56      public Transform getTarget()
57      {
58              return target;
59      }
60
61      // Returns true if targetTransform is within sight of current transform
62      public bool withinSight(Transform targetTransform, float
            fieldOfViewAngle)
63      {
64              Vector3 direction = targetTransform.position - transform.
                    position;
65              // An object is within sight if the angle is less than field of
                    view
66              return Vector3.Angle(direction, transform.forward) <
                    fieldOfViewAngle;
67      }
```

```
68 }
```

B.2 MoveTowardsLocation.cs

```
1  using UnityEngine;
2  using System;
3  using BehaviorDesigner.Runtime;
4  using BehaviorDesigner.Runtime.Tasks;
5
6  public class MoveTowardsRoom : BehaviorDesigner.Runtime.Tasks.Action
7  {
8          public SharedTransform moveTowardsNew;
9          public SharedBool canIMoveTowardsRoom = true;
10         public string messageToConsole = "Reached destination. Continuing...";
11
12         public override TaskStatus OnUpdate()
13         {
14                 if (canIMoveTowardsRoom.Value == true) {
15                         if (Vector3.SqrMagnitude (transform.position -
                               moveTowardsNew.Value.position) < 0.5f) {
16                                 Debug.Log (messageToConsole);
17                                 return TaskStatus.Success;
18                         }
19
20                         UnityEngine.AI.NavMeshAgent agent = GetComponent<
                               UnityEngine.AI.NavMeshAgent> ();
21                         agent.destination = moveTowardsNew.Value.position;
22                         return TaskStatus.Running;
23                 } else {
24                         Debug.Log ("This node is disabled. We will continue with
                               our next  task...");
25                         return TaskStatus.Failure;
26                 }
27
28         }
29 }
```

C Ontology classes

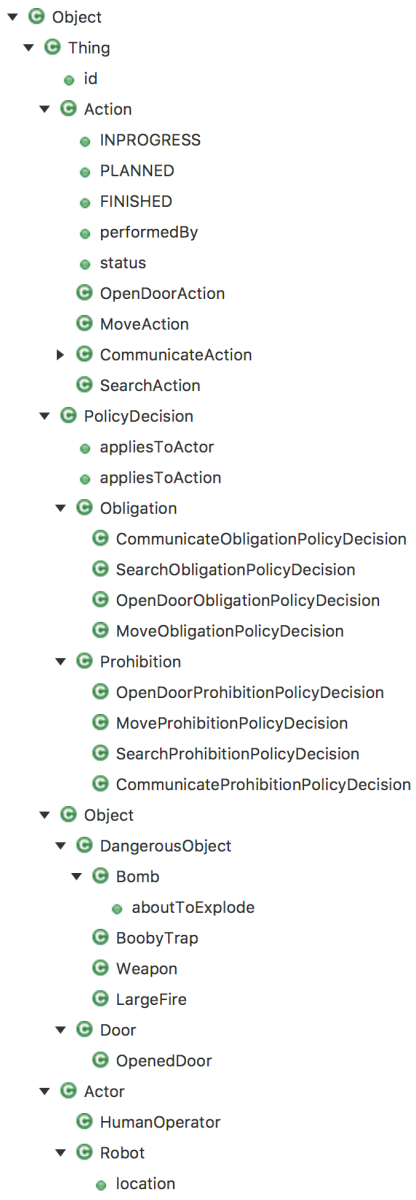


Figure 33: Ontology - Tree View

D PETUI

D.1 PETUI - policies

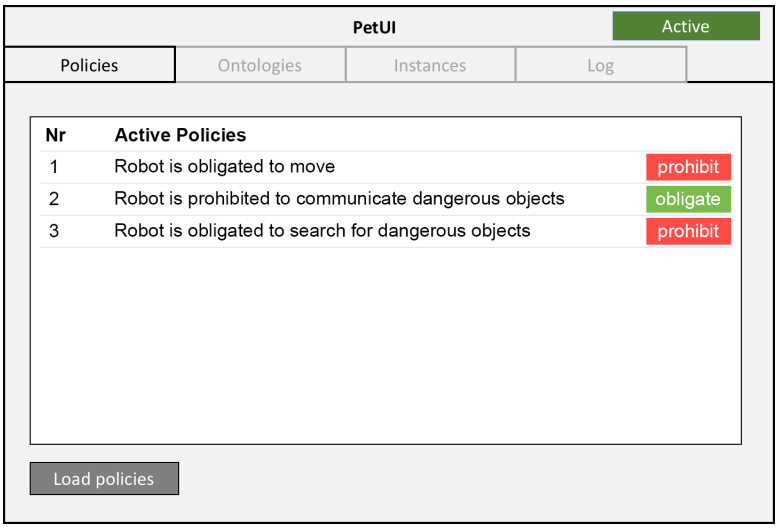


Figure 34: PETUI - policies

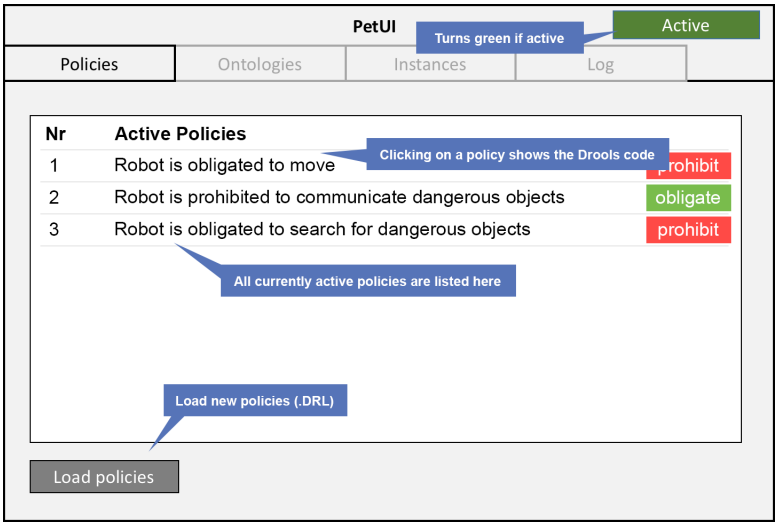


Figure 35: PETUI - policies explained

D.2 PETUI - ontology overview

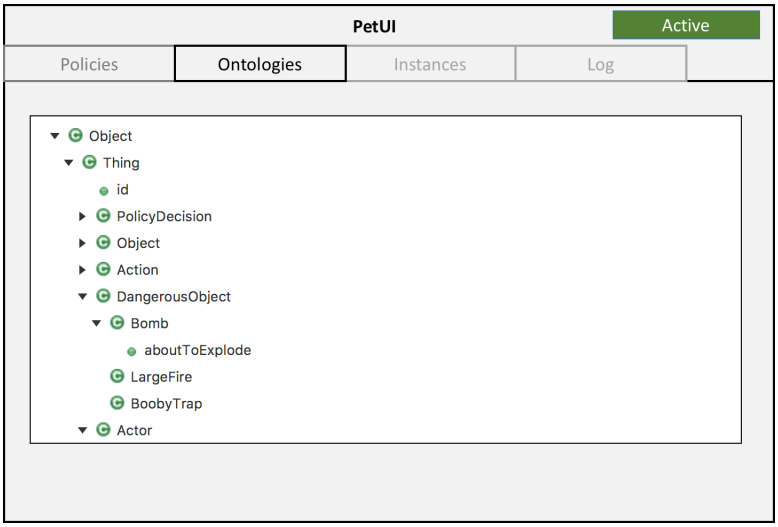


Figure 36: PETUI - ontology overview

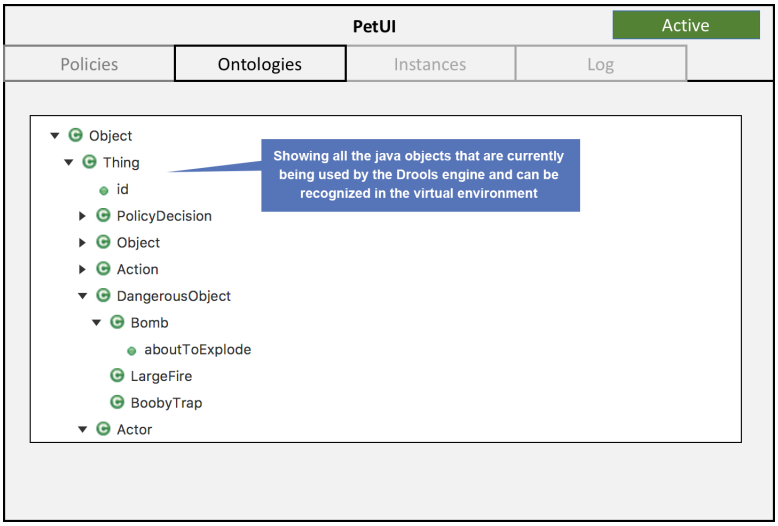


Figure 37: PETUI - ontology explained

D.3 PETUI - Instances

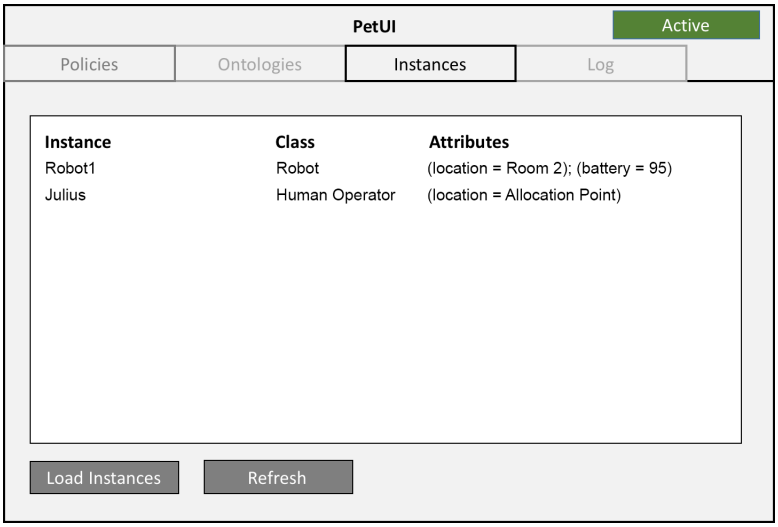


Figure 38: PETUI - instances

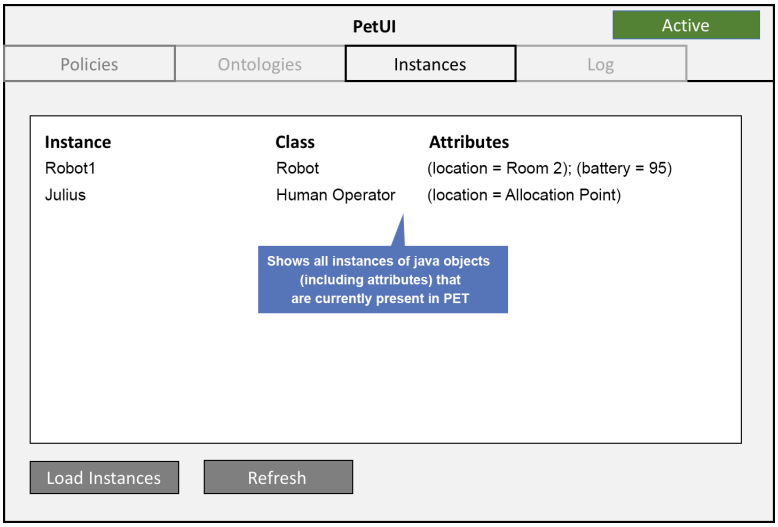


Figure 39: PETUI - instances explained

D.4 PETUI - Log

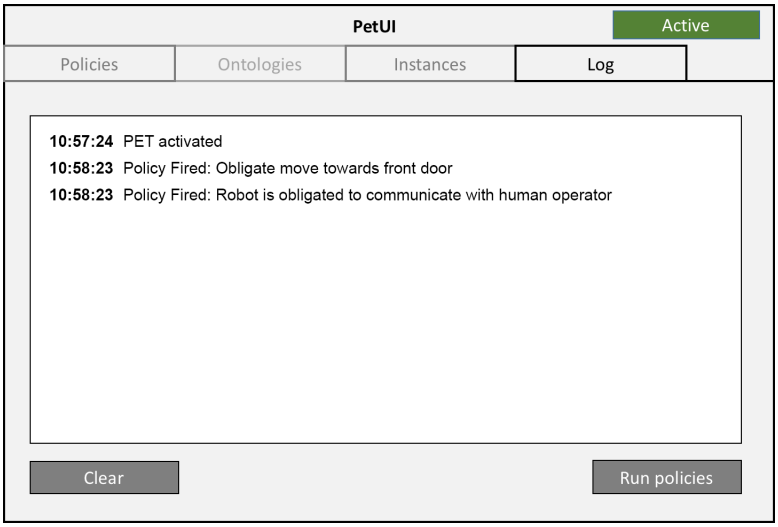


Figure 40: PETUI - log

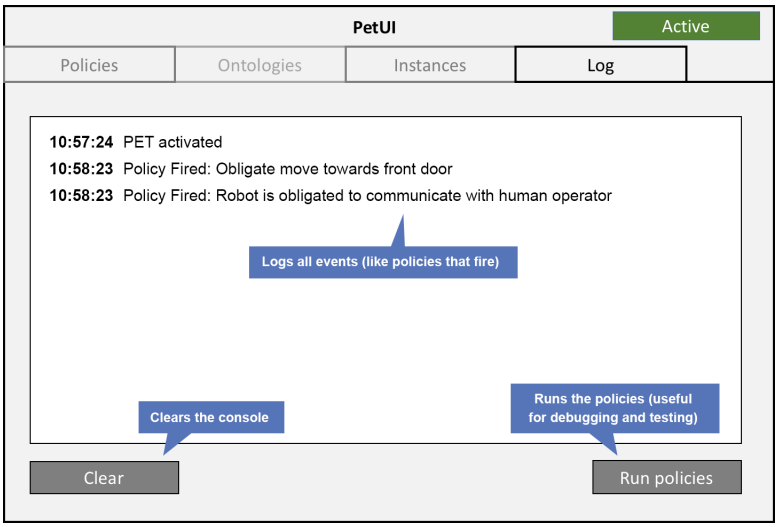
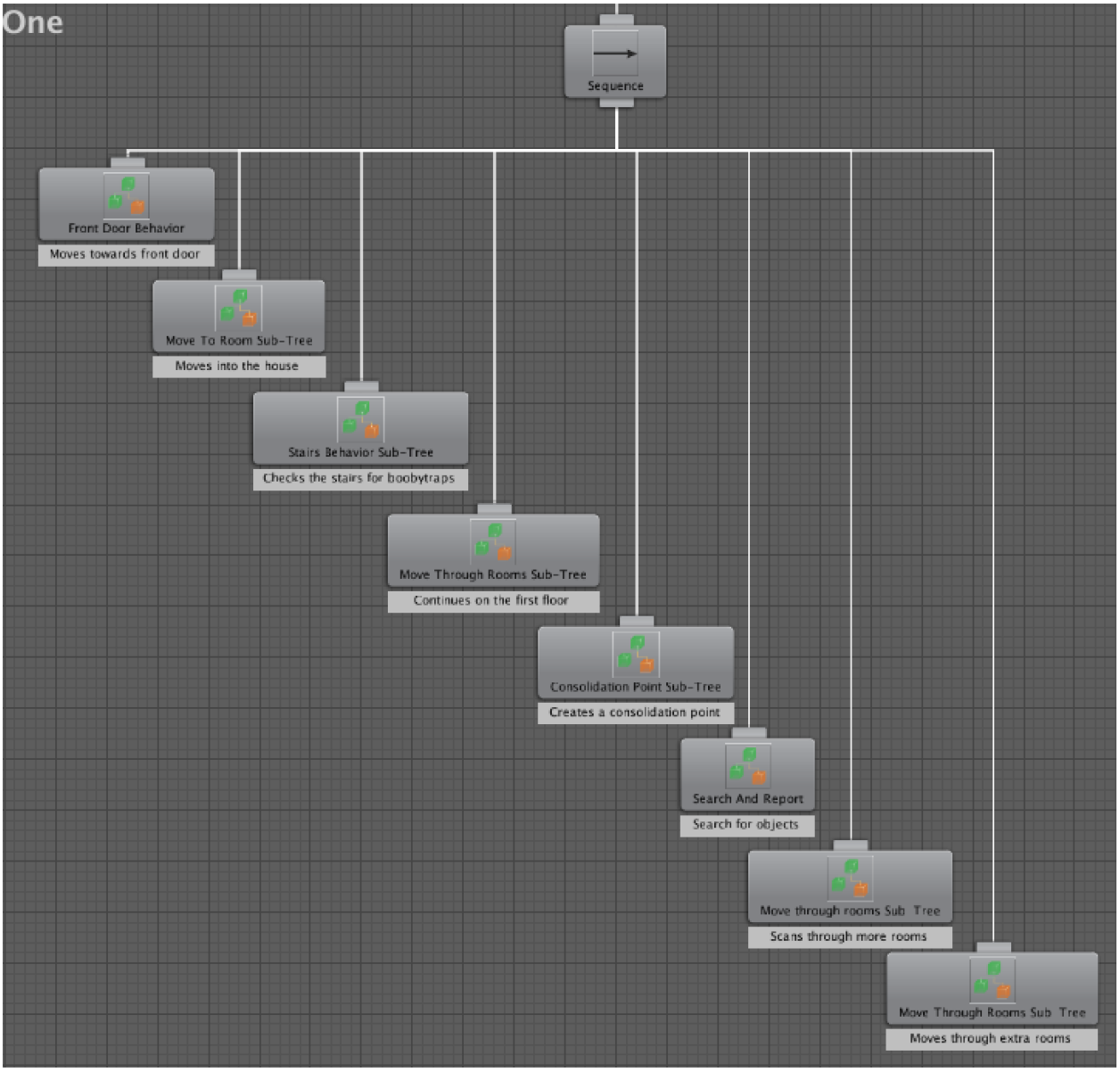


Figure 41: PETUI - log explained

E Behavior Designer - Complete Behavior Tree



References

Abramson, S., Levin, S., & Zaslavsky, R. (2006). *Robotic vacuum cleaner.* Google Patents. (US Patent 7,079,923)

Bagnell, J. A., Cavalcanti, F., Cui, L., Galluzzo, T., Hebert, M., et al. (2012). An integrated system for autonomous robotics manipulation. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on* (pp. 2955–2962).

Bradshaw, J. M., & Montanari, R. (2014). Policy-based governance of complex distributed systems: What past trends can teach us about future requirements.

Champanard, A. (2012). Understanding the second-generation of behavior trees. *Tillgänglig på*.

Colledanchise, M., Marzinotto, A., & Ögren, P. (2014). Performance analysis of stochastic behavior trees. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on* (pp. 3265–3272).

Colledanchise, M., & Ögren, P. (2017). *Behavior trees in robotics and AI, an introduction* [Online]. Available from http://arxiv.org/abs/1709.00084.

Fensel, D. (2001). Ontologies. In *Ontologies* (pp. 11–18). Springer.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, *19*(1), 17–37.

Gershenson, J., Prasad, G., & Zhang, Y. (2003). Product modularity: definitions and benefits. *Journal of Engineering Design*, 295-313.

Kagal, L., Finin, T., & Joshi, A. (2003). A policy language for a pervasive computing environment. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on* (pp. 63–74).

Klingspor, V., Demiris, J., & Kaiser, M. (1997). Human-robot communication and machine learning. *Applied Artificial Intelligence*, *11*(7), 719–746.

Klöckner, A. (2013). Behavior trees for UAV mission management. *INFORMATIK 2013: Informatik angepasst an Mensch, Organisation und Umwelt*, 57–68.

Martius, G., Der, R., & Ay, N. (2013). Information driven self-organization of complex robotic behaviors. *PloS one*, *8*(5), 363-400.

Marzinotto, A., Colledanchise, M., Smith, C., & Ögren, P. (2014). Towards a unified behavior trees framework for robot control. In *2014 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 5420–5427).

Nof, S. Y. (1999). *Handbook of industrial robotics* (Vol. 1). John Wiley & Sons.

Ogren, P. (2012). Increasing Modularity of UAV Control Systems Using Computer Game Behavior Trees. In *Aiaa guidance, navigation, and control conference* (p. 4458).

Paxton, C., Hundt, A., Jonathan, F., Guerin, K., & Hager, G. D. (2017). CoSTAR: Instructing collaborative robots with behavior trees and vision. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on* (pp. 564–571).

Peeters, M., Van Den Bosch, K., Meyer, J.-J. C., & Neerincx, M. A. (2012). An ontology for integrating didactics into a serious training game. *CEUR Workshop Proceedings*, *898*, 1–10.

Prakken, H., & Sartor, G. (2015). Law and logic: a review from an argumentation perspective. *Artificial Intelligence*, *227*, 214–245.

Richardson, D. E. (2001). *Robot policies for monitoring availability and response of network performance as seen from user perspective.* Google Patents. (US Patent 6,317,788)

Salatino, M., De Maio, M., & Aliverti, E. (2016). *Mastering JBoss Drools 6.* Packt Publishing Ltd.

Susi, T., Johannesson, M., & Backlund, P. (2007). *Serious games: an overview.* Institutionen för kommunikation och information. Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-1279.

Thulasiraman, K., & Swamy, M. (1992). 5.7 Acyclic Directed Graphs. *Graphs: Theory and Algorithms*, 118.

Witten, I. H., Frank, E., Hall, M. A., & Pal, C. J. (2016). *Data mining: Practical machine learning tools and techniques.* Morgan Kaufmann.