Towards Reactive and Robust Manipulation Tasks using Behavior Trees

Michele Colledanchise and Lorenzo Natale

Abstract— **The applications of manipulation tasks will soon move from caged robots in assembly lines to the more challenging household and collaborative ones that have to operate in dynamic and unpredictable environments. Due to the uncertainty of such environments, it is preferable to perform tasks in a reactive and robust fashion. Classic task execution architectures fail to do so in reasonable complexity, which suggests considering alternative execution architectures. In this paper, we outline how to achieve reactive and robust manipulation by encoding the task as a Behavior Tree (BT). To demonstrate our approach, we illustrate an example of a manipulation task and how the BT deals with external disturbance and failures at the task level. We provide a preliminary comparison with Finite State Machines and State Charts.**

I. Introduction

To date, robots have been very successful at manipulating objects in fixed and structured environments where the robot and the surroundings are predictable in space and time. While designing the robots software, engineers know what to expect, and when. However, factories will soon have humans and robots working closely together. The introduction of a human worker in the robot's environment results in the loss of the environment's predictably: the worker can pick the object that the robot planned to pick or they can move a tool in a position that is not reachable by the robot. Household robots will manipulate a large variety of very different objects and have more complex but less precise end-effectors than the ones in factory lines. As a consequence, many manipulation tasks may fail and need to be re-executed. Hence, outside the typical factory lines, the manipulation tasks need to be designed to be reactive and robust. By reactive we mean the capability to react to an unexpected exogenous event (e.g. an object that slips out from the robot's hand), by robust we mean the capability to operate properly in case of failures at task level (e.g. an arm in not usable is a certain situation).[1] Classical mathematical models for task execution as Finite State Machines (FSMs) or State Charts fail to represent a reactive and robust task in a compact and human-readable manner. In particular, FSMs need to have many outgoing transitions from an action, each of which models a possible event the robot has to react accordingly; State Charts need a manually-designed hierarchy to reduce the number of transition. For this reason, we propose Behavior Trees (BTs) as an alternative model to represent task execution.

The authors are with the iCub Facility, Istituto Italiano di Tecnologia. Genoa, Italy. e-mail: michele.colledanchise@iit.it

[1]We use the term robustness aligned with the IEEE standard glossary of software engineering terminology: "The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions."

BTs are a graphical mathematical model for reactive fault tolerant task executions first introduced in the computer gaming industry to control in-game opponents [1], and is now an established tool used in the computer game community. BTs are appreciated for being human-readable, highly modular, flexible and reusable, and have also been shown to generalize other successful control architectures such as the Subsumption architecture and the Teleo-reactive Paradigm [2].

II. Behavior Trees

In this section, we briefly describe the semantic of BTs. A more detailed description can be found in [2], [3].

A BT is a graphical modeling language used as a representation for execution of observation-based actions in a system. A BT is represented as a rooted tree where the internal nodes are called control flow nodes and leaf nodes are called execution nodes.

Graphically, the children of a node are placed below it and they are executed in the order from left to right, as will be explained later.

The execution of a BT begins from the root node. It sends activation signals called *ticks* with a given frequency to its children. When a parent sends ticks to a child, the execution of the child is allowed. The child returns to the parent a status *running* if its execution has not finished yet, *success* if it has achieved its goal, or *failure* otherwise.

In the classical representation, there are four types of control flow nodes (fallback, sequence, parallel, and decorator) and two execution nodes (action and condition). Below we describe the execution of the most common nodes used.

Fallback: When a fallback node receives ticks, it routes them to its own children from the left, returning success (running) as soon as it finds a child that returns success (running). It returns failure when all the children return failure. When a child i returns running or success, the fallback node does not tick the next child (if any) but keeps ticking all the child up to child i. The fallback node is graphically represented by a box with a "?", as in Figure 1.

Sequence: When a sequence node receives ticks, it routes them to its own children from the left, returning failure (running) as soon as it finds a child that returns failure (running). It returns success when all the children return success. When a child i returns running or failure, the sequence node does not tick the next child (if any) but keeps ticking all the child up to child i. The sequence node is graphically represented by a box with a "$\rightarrow$", as in Figure 1.

Action: When an action node receives ticks, it performs some operations. It returns success to its parent if the action is completed and failure if the action cannot be completed. Otherwise, it returns running. An action node is graphically represented by a rectangle, as in Figure 1.

Condition: The condition node checks if a proposition is satisfied or not, returning success or failure accordingly. A condition node is represented by an ellipse, as in Figure 1

III. Reactive Manipulation

In this section, we outline how to obtain a reactive manipulation tasks using BTs. A reactive manipulation task can be easily obtained by coupling each action of a sequence with the condition that is meant to hold in fallback.

Example 1: Consider the BT in Figure 1, executed whenever a humanoid robot has to perform a pick and place task. The root is a sequence node that ticks its first child, which is a fallback node, that again ticks its first child, the condition node *Object Close*. If the object is not close, then the condition node returns failure and the fallback node ticks its second child: the action node *Approach Object*. If the object is indeed close, then the condition node returns success, its parent (the fallback node) returns success, and the root node ticks its second child, which is a (different) fallback node. This node ticks its first child: the condition node *Object Picked*. If the object is not picked, the fallback node ticks the action node *Pick Object*. The execution continues similarly for the remaining nodes.

Note that the root node keeps sending ticks to the first child continually, and it keeps sending ticks to the second child only when the first one returns success and it keeps sending ticks to the third child if the first and the second ones return success. This allows us to execute the task in a reactive fashion. In Example 1, while the robot is performing the action *Pick Object*, the robot is still checking if the object is close or not. If the object moves away unexpectedly (e.g. a human operator moves the object) then the robot aborts the execution of *Pick Object* and starts the execution of *Approach Object*. Similarly, the execution of *Place Object* is aborted if the object slips out from the robot's hand.

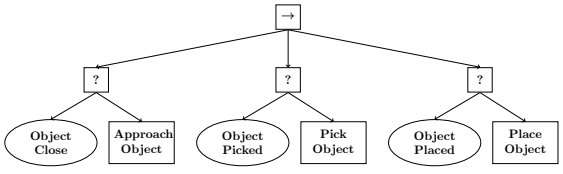

Fig. 1. Example of a reactive manipulation task encoded as a BT.

IV. Robust Manipulation

In this section, we outline how to obtain robust manipulation tasks using BTs. A robust manipulation task can be easily obtained by adding the different options for achieving the same task (e.g. right or left hand grasp) in fallback.

Example 2: Consider the BT in Figure 2, executed whenever a humanoid robot has to grasp an object. The root is a fallback node that ticks its first child: the action *Right Hand Grasp*. If the action fails, then the root node ticks its second child: the action *Left Hand Grasp*. If also this action fails, then the root node ticks its third and last child: the action *Both Hands Grasp*.

Example 2 highlights how the modularity of BTs allows us to separate the different grasping routines and to execute them in a fallback fashion.

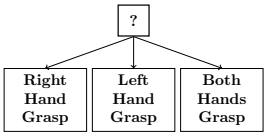

Fig. 2. Example of a robust manipulation task encoded as a BT

V. Preliminary Comparison

To truly understand the advantages of the peculiar structure and execution of BTs, we compare BT with a FSM and a State Charts that encode the manipulation task in Example 1.

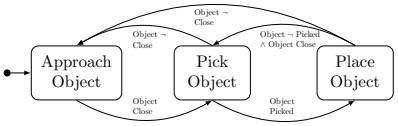

Fig. 3. Example of a reactive manipulation task encoded as a FSM

Figure 3 shows the FSM that encodes the task. To obtain a reactive behavior, the FSM needs to have a transition from each action, to an action that was supposed to be done. Moreover, all the outgoing transitions of an action need to be mutually exclusive, which increases the number of transitions needed.

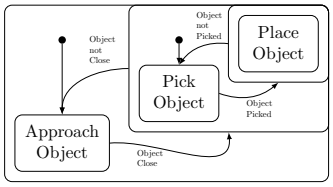

Fig. 4. Example of a reactive manipulation task encoded as a State Chart

Figure 4 shows the State Chart that encodes the task. To obtain a reactive behavior, the state chart needs to have a hierarchy defined manually for each action, which makes it less intuitive and unreasonably intricate.

VI. Conclusions

In this paper, we outlined how to obtain a reactive and robust manipulation task using BTs. The resulting BTs are modular and compact, which makes them easier to model compared with classical approaches as FSMs or State Charts.

References

[1] D. Isla, "Handling Complexity in the Halo 2 AI," in *Game Developers Conference*, 2005.

[2] M. Colledanchise and P. Ögren, "Behavior Trees in Roboics and AI: An Introduction," *CoRR*, vol. abs/1709.00084, 2017. [Online]. Available: http://arxiv.org/abs/1709.00084

[3] P. Ögren, "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees," in *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.