Hindawi
Journal of Robotics
Volume 2022, Article ID 3314084, 9 pages
https://doi.org/10.1155/2022/3314084

Review Article

Behavior Trees for Smart Robots Practical Guidelines for Robot Software Development

Eric Dortmans and Teade Punter

Fontys University of Applied Sciences, Research Group High Tech Embedded Software, Eindhoven, Netherlands

Correspondence should be addressed to Teade Punter; teade.punter@fontys.nl

Received 20 May 2022; Revised 1 July 2022; Accepted 23 July 2022; Published 7 September 2022

Academic Editor: Ramadoni Syahputra

Behavior Trees are a promising approach to model the autonomous behaviour of robots in dynamic environments. Behavior Trees represent action selection decisions as a tree of decision nodes. The hierarchy of these decision nodes provides the planning of actions of the robot including its reactions on exceptions. Behavior Trees enable flexible planning and replanning of robot behavior while supporting better maintainable decision-making than traditional Finite State Machines. This paper presents an overview of lessons, which we have learned when applying Behavior Trees to various autonomous robots. We present these lessons as a sequence of steps that is meant to support robot software practitioners to develop their systems.

1. Introduction: Need for Embodied Intelligence

Robots are designed to get a particular task done in the real world such as manoeuvre along shop floors in warehouses, or to support caring staff in hospitals. They are made to perceive the world using specific sensors, such as camera's, laser scanners, pressure sensors, and to act in this world using specific actuators like grippers, arms, and wheels [1].

In general, an autonomous robot has a control unit that takes sensor data as input, builds a model of the world, and selects appropriate actions for the situation it thinks it is in. To act in its world the robot has a set of built-in *skills* [2], such as the ability to move around and to pick and place objects. (see Figures1 and 2)

The decision-making intelligence of an autonomous robot is in its action selection mechanism that determines which skill to activate for which state of the world. The action selection mechanism can be seen as the centre of the robot brains. It determines its behavior and with that its usefulness.

Designing a robust and maintainable action selection policy is not an easy task. Not only the primary task of the robot needs to be encoded as a sequence of actions but also all possible real-world states must also be taken into account. In general, a policy will comprise a combination of a deliberative plan of actions and a set of actions that are triggered only when the state of the world requires it.

As the real world is too complex to deal with in full detail, it is common practice for roboticists to make abstractions of it. Every detail that is not relevant for the task at hand is abstracted away. In this paper, we will use the following abstractions:

(i) We represent the state space of the world using a collection of boolean conditions. Each condition is a predicate that the robot can check for its truth value. In this way, we basically divide the complex and analogue state-space of the world into a limited amount of discrete states.

(ii) Complex actuation algorithms ("skills") of the robot are abstracted as a collection of actions *A* that can bring the world (with high probability) from a precondition to a postcondition.

(iii) We assume that the conditions can be checked instantaneously while actions may take some (unknown) time to complete.

(iv) The mapping of states S to actions A is represented as a so-called action selection policy $\pi: S \longrightarrow A$.

Action selection policies can be implemented in different ways. A traditional way to implement such a policy is by using imperative programming language constructs, such as statement sequences, if-then-else, and while constructs. The example in Figure 3 shows a procedural policy, specified using pseudocode, for an industrial robot that has to move objects.

The example robot has four actions that can be selected: "move_to_object," "grab_object," "move_to_goal," and "release_object." Something might go wrong, and the robot should "stop." This results in a fairly simple flow depicted in the left part of the figure. However, things get complicated when sensor inputs are taking into account as well, see right part of Figure 3.

In general, action selection policies are strongly influenced by the environment in which the robots have to work. Modern, autonomous robots usually have to operate in unstructured environments and have lots of sensors to deal with. Coding an action selection policy in the traditional, procedural way leads to complex code that is hard to develop, maintain, and improve.

Therefore, we prefer to model robot behavior explicitly instead of implicitly coding it in some programming language. Our preferred way of modelling is using Behavior Trees (BTs) [3, 4]. BTs generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees [5]. BTs are an efficient as well as flexible way of creating complex systems that are both modular and reactive. These properties are crucial in many applications, which have led to the spread of BTs from computer game programming to many branches of AI and Robotics [6]. Several authors have already shown that Behavior Trees are a promising approach to model autonomous behaviour in dynamic environments [6–11], [25, 27].

A growing amount of literature discusses fundamental aspects and applications of behavior trees for robotics [3, 6, 7]. Several authors provide formal definitions of Behavior Trees [3, 4] or propose extensions [12]. Others provide algorithms to implement goal-directed task plans using behavior trees and provide convergence proofs [13–17].

The goal of this paper is to bridge the gap between fundamental research and practical application of Behavior Trees for action selection in autonomous robots. We provide guidelines how to construct BTs to implement reactive, robust, and maintainable robot behavior. We also provide advice how to build a flexible BT execution infrastructure.

Section 2 introduces the BT formalism and its main syntax. Section 3 provides an overview of advantages and disadvantages of using BTs. The remaining sections provide lessons we learned during BT application (Section 4) and are aimed to help robot software engineers in building a flexible execution architecture for BTs (Section 5).

2. BT Syntax and Semantics

A Behavior Tree [3] describes the decision-making policy of a robot (or software agent or non-player game character) as a rooted, directed tree of nodes. The tree consists of flow

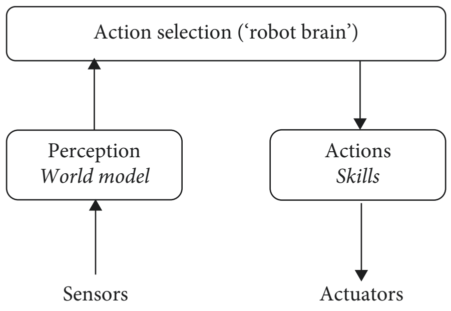


Figure 1: Robot software architecture embodying intelligence: perception, action, and world model.

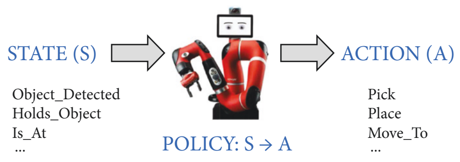


Figure 2: Action selection.

control nodes, that represent the decisions in the tree, and leaf nodes that are the actions to be executed and the conditions under which those actions are executed.

Four different flow control types are distinguished:

(i) Sequence ($\longrightarrow$)—Execute child nodes one by one until one fails. Execute the first child, and then the next one until all children have succeeded.

(ii) *Fallback* or *Selector* (?)—Execute child nodes one by one until one succeeds.Try the first child, and if it does not succeed then try the next one until one of the children succeeds.

(iii) *Parallel* (||)—Execute all child nodes in parallel until a specified number (e.g., all) of them succeed.

(iv) *Decorator* (◊)—Modify the execution (result) of its (only) child node. For example: invert, time-out, and repeat-until-success.

Leaf nodes represent the world conditions that are to be checked and the actions that can be selected:

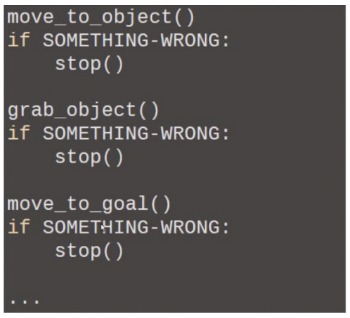
(i) *Condition*– Check the state of the world (model).

(ii) *Action*– Act upon the world using a particular skill.

In practice, most BT libraries have all kinds of variations of the above introduced flow control nodes and also allow to extend the basic node set with domain-specific nodes [7]. Action and condition nodes are mostly domain and application specific.

Figure 4 shows an example of a simple, intuitive BT of a robot that has to pick an object, move it to a destination, and place the object at that location.

The action selection policy of the robot is fully defined by the structure and semantics of the flow control and leaf nodes of the BT. The hierarchy of decision nodes concisely specifies all possible sequences of actions of the robot, and how these sequences are influenced by conditions.

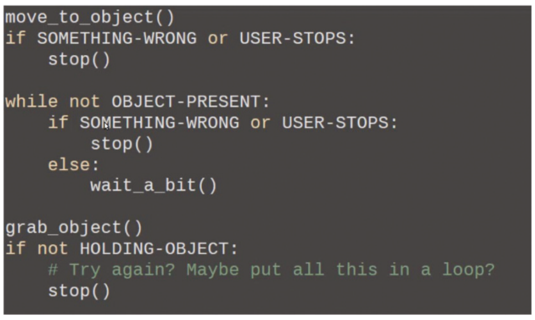


```
move_to_object()
if SOMETHING-WRONG:
    stop()

grab_object()
if SOMETHING-WRONG:
    stop()

move_to_goal()
if SOMETHING-WRONG:
    stop()

...
```



```
move_to_object()
if SOMETHING-WRONG or USER-STOPS:
    stop()

while not OBJECT-PRESENT:
    if SOMETHING-WRONG or USER-STOPS:
        stop()
    else:
        wait_a_bit()

grab_object()
if not HOLDING-OBJECT:
    # Try again? Maybe put all this in a loop?
    stop()
```

Figure 3: Example of a procedural policy: basic code (left), extended with sensor input (right).

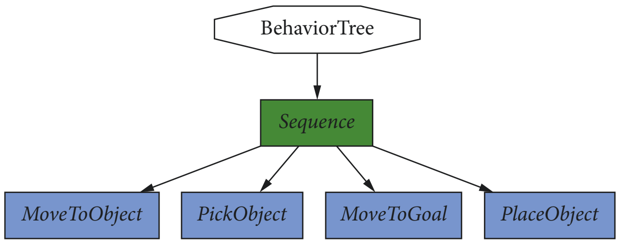


Figure 4: Naïve BT with four action nodes and a "sequence" control flow node.

To be able to execute a BT, each node supports a "tick" method. When this method is called (by its parent node), the node is activated. If a node needs more time than one tick to execute, the tick method returns status "RUNNING" to its parent. When the node completed its task, its tick method returns "SUCCESS" if its outcome was positive or "FAILURE" otherwise.

A BT is executed by calling the tick method of the root node at a certain frequency. Ticking the BT needs to continue while its root node returns "RUNNING" and stop when the root node returns SUCCESS or FAILURE.

Each flow control node passes the tick on to its child nodes, one by one, or in case of a parallel node all at the same time. Eventually the tick reaches a leaf node. The tick allows this leaf node to execute itself, i.e., to check a condition, or to apply a skill. A flow control node that receives a status "RUNNING" from a child will return "RUNNING" to its parent. How it reacts to receiving "SUCCESS" or "FAILURE" status from a child depends on its type.

It is important to note that Sequence and Fallback nodes tick their children one by one. This automatically implies a priority: the first child (from the left) is run before the second child and so on. This can be used to give certain actions higher priority than others.

A Fallback node works like a logical "OR" function, but instead of checking the status of its children all at once, it ticks its children one by one, starting with the first, left-most child. Fallback is often used for trying prioritized alternatives that, when executed, have the same effect, i.e., would result in the same postcondition.

A Sequence node works like a logical "AND" function, but again checking its children one by one instead of all at the same time. A Sequence enforces an all or nothing transaction. It can be used to run a sequence of actions as a transaction, or to always check a precondition before starting an action.

3. BT Trade-Offs

Behavior Trees provide an alternative to Hierarchical State Machines (HSMs). Both are equally expressive. We perceived the following advantages when applying BTs [3]:

(i) Modularity, Reusability—BTs are modular on all scales. Any subtree from a BT or even any complete BT can be reused as a subtree in another BT.

(ii) Readability—Their tree structure and modularity makes BTs well readable and therefore better analysable and understandable by humans.

(iii) Maintainability—The tree structure of BTs is better maintainable than the transitions in HSMs. Decisions in BTs are represented explicitly as nodes, and are not hidden in state transition conditions.

(iv) Learnability—BTs can be learned by (machine) learning programs [3] or generated from a formalized plan [15].

On the negative side, we should mention:

(i) Verifiability- BTs are not yet supported by formal verification tools, contrary to HSMs. This shortcoming can, however, be addressed by translating the BT into an HSM and then conducting the verification, see, e.g., the Carve project (https://carve-robmosys.github.io/results/).

4. A Practical Approach to Define BTs

This section aims at providing a structured approach for robot practitioners for constructing robust and reactive BTs. We will describe four steps. These steps result from our lessons learned when applying BTs in robot research projects at Fontys University of Applied Sciences. The steps we will elaborate on are:

(1) Describe context and purpose of the robot.

(2) Basic plan—describe a "good weather" BT.

(3) Create a robust plan.

(4) Add reactivity to the plan.

Specific idioms, needed to go from step 2 to 3, are introduced as an intermezzo. We illustrate the steps by means of a running example. The example that we have chosen is a mobile robot that picks up objects and places them at some goal location.

4.1. Step 1: Describe Context and Purpose of the Robot. Before even starting to create a BT, a context description of the robot application helps to identify the relevant aspects to be covered by your BT. Thinking about the system context first is a general good design principle, which is, e.g., advocated by the C4-model (https://c4model.com/). The following aspects are helpful when defining the context for a robot application: system goal, actions, scenarios, and states; inspired by Winikoff and Padgham [18].

The *system goal* describes what the robot should do. The goal for our robot is moving objects to a goal location. This should start the thinking process. What actions does the robot need to perform? Which world and robot states should we be able to check? What action to execute in which state?

The *actions* describe which action skills are needed by the robot to achieve its system goal? In case of our example robot, obvious actions will be "move to object," "pick object," "move to the goal," and "place object." In reality, there are usually many more actions to consider.

A *scenario* describes the dynamics of the robot, by linking the actions into a sequence, like first moving to an object object, then picking it up, then moving to the goal location, and finally placing it there. Although writing down a scenario might look straightforward, in practice often problems pop up when you think about all situations that can interrupt the normal sequence of actions, e.g., when our robot finds itself at a closed door or when its battery is empty. Do always take non-functional requirements, like power or safety, into account when thinking about scenarios. Being aware of contingencies, and thinking in terms of "bad weather" behavior for the robot, does in general help to define a robust BT (see step 3). In general the "good weather" scenario is just one of the scenarios to consider.

States are related to the scenarios because they define when actions are eligible for execution. In our example, the most important state of the robot is its position in the world. Dependent on its position (at origin, at goal), appropriate actions can be conducted, like: picking at origin, placing at goal. In general, there are many more states to consider than just the robot position, such as the location of the objects, the state of the robot gripper, or the state of the robot battery. Being aware of which states are relevant helps to refining the BT (see step 2).

4.2. Step 2: Basic Plan, Define a "Good Weather" BT. When the context and purpose of the robot are defined, it is time to draw a first BT to describe the task that the robot has to do as a sequence of actions. This procedure should result in a first, intuitive version of the BT which should work when there are no disturbances, a so-called "good weather" BT.

In our running example, our BT is identical to the BT of Figure 4, a simple sequence of actions: "MoveToGoal," "PickObject," "MoveToGoal," and "PlaceObject." This is the same sequence that we would write down as a programmer. Nothing gained yet by using a BT. However, the "good weather" scenario is only valid under the following assumptions:

(1) The state of the world at the start is as required by the scenario.
(2) All actions do eventually complete successfully.
(3) No external agent does interfere by changing the world state, while the scenario is running.

In practice, it is unlikely that all these assumptions hold true; in our example:

(1) There might not be an object at the pickup location.
(2) Grabbing the object might fail.
(3) The robot might lose the object it has picked.
(4) The robot might run out of (battery) power.
(5) An external agent, such as a human or another robot, might take the object away and puts it at the goal location.

This implies that many more possible scenarios should be incorporated in our BT. In general, a good BT should be a concise model that describes all the possible scenarios, i.e., all possible sequences of actions. In order to select between various scenarios, we need to add conditions to the BT, i.e., we need to check the actual state of the world (including the robot itself) to decide which action is eligible for execution.

4.3. Intermezzo: Adding Checks (Idioms) for Robustness. The BT that was derived in step 2 encodes a "good weather" scenario and assumes that all actions go well. It does not react on any disturbances and just gives up if one of the actions fails. In the previous step, the robot is modelled by ordering the actions as *then* . . . *else*, but not taking the *if*, the condition, really into account. These *ifs*, the conditions or guards, are added to ensure the appropriate context to execute the action. We can do better than that by adding checks to enrich the BT with other possible ways to execute to make the BT more robust.

Several checks can be added. A precondition check can be added to the action (or subtree) using a Sequence node, as depicted in Figure 5 (left). A postcondition check can be added using a Fallback node as depicted in Figure 5 (right).

A specific pattern that is constructed from these idioms is the postcondition precondition action (PPA) pattern [3]. Several actions can have the same postcondition but with different preconditions. By combining these alternatives using a Fallback node, we can improve the robustness of the BT. The PPA pattern is a subtree which combines fallback to alternative actions with precondition and postcondition checks. (see Figure 6)

4.4. Step 3: Create a Robust Plan. Now, the idioms—to achieve robustness—have been introduced, we can apply them. Step 3 is about the creation of a robust BT, to execute a

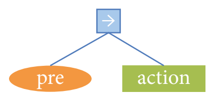
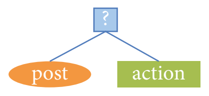
	
Precondition–specifies what must be true to execute a skill. It is the condition that applies to a sequence of skills. In pseudocode: if pre do action.	Post condition–specifies what will be true when a skill is executed. It applies to a fallback of skills: if the condition is not fulfilled another action should be done. In pseudocode: if not post do action.

FIGURE 5: Pre- and post-condition defined.

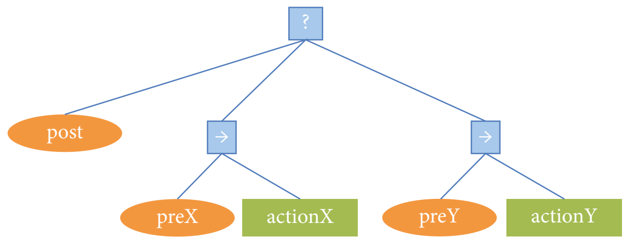


FIGURE 6: Postcondition-precondition-action (PPA) pattern subtree.

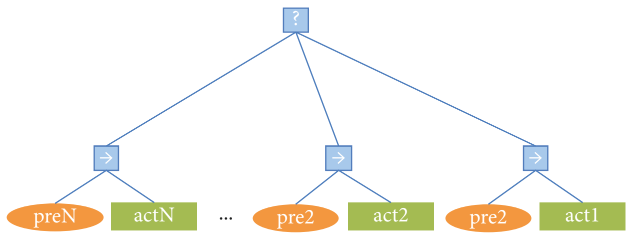


FIGURE 7: Representation of a robust logical-dynamical chain (RLDC).

deliberate, goal-oriented plan in an effective and efficient way. The way to generate such a plan in the form of a BT can be done manually as well as automatically by a deliberative planner.

A straight forward way to define a robust BT is by using the "Robust Logical-Dynamical Chain" pattern [17] Figure 7, which is also described as "Implicit Sequence" in [3]:

(1) Reverse the order of the actions, defining most downstream action first (reading from the left of the tree).

(2) Replace the Sequence node by a Fallback node in order to change from an explicit to an implicit sequence.

(3) Add a precondition check to each action.

When we apply this pattern on our running example, we reverse the order of the actions, compared to how they were defined in the naïve BT or basic plan BT, Figure 4. The result is shown in Figure 8.

If the precondition of an action fails, we fall back to the next (more upstream) action, and so on. The order of the actions is reversed because we are primarily interested in running the last, most downstream action, because running that action will give us the outcome we want.

Note that in this case a so-called "ReactiveFallback" node is used. This node always ticks all its children (from left to right), even those that have already returned "SUCCESS" during earlier ticks.

An alternative approach to create a deliberative plan is expanding the BT by using "backward chaining" [13, 14, 16]. The method here is to iteratively expand preconditions. We start by taking the last, most downstream action, and work our way backwards. The precondition of this action is replaced with the PPA subtree which postcondition is the same as this precondition and so on. (see Figure 9)

When applying backward chaining to our running example, the result will be the following robust BT Figure 10.

Martín et al. [15] describe yet another approach to generate a deliberate BT. They start by generating a plan using a PDDL-based planner [19] and then automatically convert this plan into a robust BT for execution.

4.5. Step 4: Add Reactivity to the Plan. The deliberative plan for the robot ensures that the robot will do its primary task. However, there might be contingencies to deal with. A robot can only fulfill its task when it remains safe and in good working condition. While it is executing its deliberate plan, problems may occur, like an empty battery. We do not know if and when these problems will occur, but we should take care of them as soon as possible. Handling contingencies is of higher priority than executing a deliberate plan. The Fallback node already handles this priority. Its leftmost child is ticked first and thus has the highest priority. From left to right, priority goes down. Contingencies therefore have to be added to the left of the existing deliberative subtree as depicted in Figure 11.

In our running example, we introduce reactivity to the deliberative plan by adding a check on battery power level of the robot. If the power level will drop below a minimum level, the robot is triggered to move to its charging station and starts charging itself. This functionality is added as rows 3 and 4 in Figure 12, which is a supplement on the left compared to the deliberate BT that was earlier created in Figure 10. The resulting Figure 12 presents a robust and reactive BT.

5. BT Execution Architecture

Now, the basic steps for creating a robust BTs have been introduced and the BT-models have to be executed by real code. This section deals with the construction of a BT

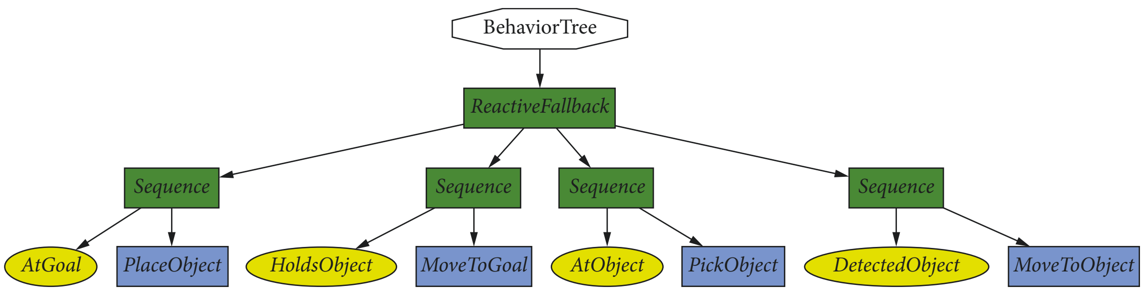


Figure 8: Result of step 3—robust logical-dynamical chain.

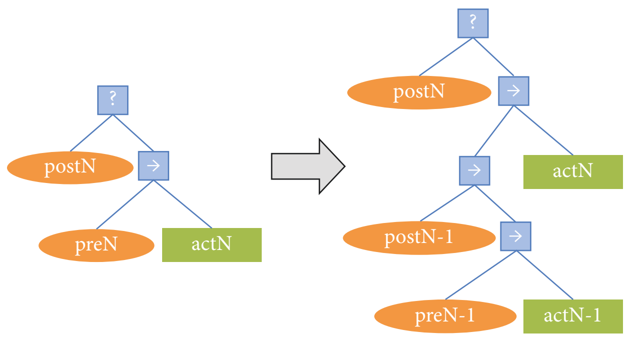


Figure 9: Expanding the BT using backward chaining.

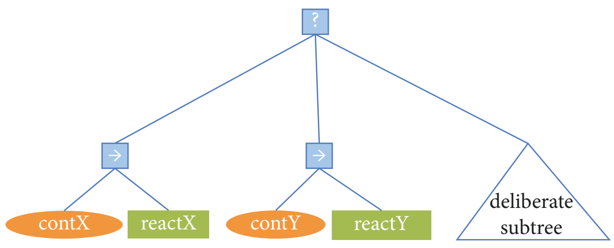


Figure 11: Handling contingencies by adding reactivity to the plan.

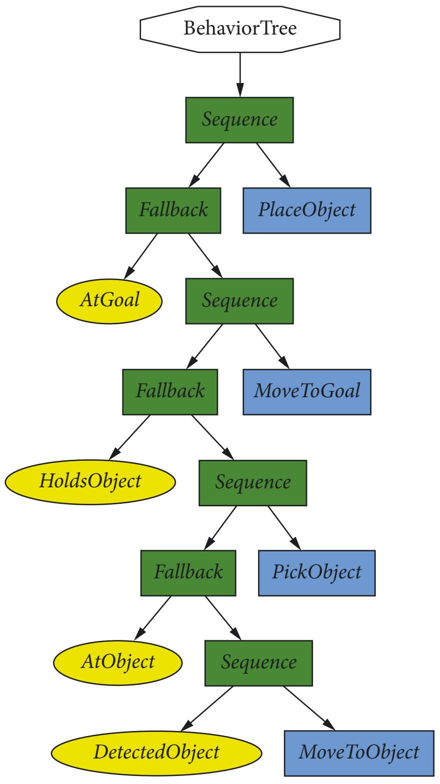


Figure 10: Another result for step 3—by backward chaining.

execution architecture for the successful application of BTs in robots.

Robot software is typically divided into three tiers [20], comprising a deliberator (planner) tier, a task executor (sequencer) tier, and a controller (skills) tier. The deliberator produces high-level task plans. The executor is responsible for the execution of those plans, by sequencing primitive skills. In a reactive robot architecture [8, 21, 22], the executor is in charge of driving the overall system by requesting new plans from the deliberator and dispatching subtasks to specific robotic controller (skill) modules. In this case, the executor considers the deliberator as one of the available skills.

The three-tier architecture can be applied at multiple abstraction levels. Some skills might be quite complex and will require a three-tier structure themselves. For instance, a "NavigateTo" skill needs a motion planner to calculate a path in the map and a controller to follow that path while avoiding collisions and a recovery skill that gets the robot out of navigation problems. A BT-based executor could be used to coordinate these subskills [8].

A deliberator is essential in applications where plans need to be constructed at run time. Runtime planning takes time but does also provide optimized and efficient behavior. For a lot of applications, a good enough plan can be constructed at design time, either manually or using an automatic planner. This holds in particular for reactive, behavior-based robotic applications [23]. Design time generated plans have the advantage that they can be validated or verified before deployment. Flexibility can be added to such a plan by runtime switching between predefined subplans.

Skills [2, 12, 24] are basic building blocks of task plans. A robotic manipulator arm typically has skills like "Pick" and "Place" to be able to pick-up an object and place it somewhere. A mobile robot typically has skills like "NavigateTo" to plan a path and follow that path while avoiding objects. Basic skills related to object detection, navigation, and arm motion are common to a lot of robots. Added to those common skills, specific application domains like manufacturing [24] or robot soccer [25] require domain specific skills.

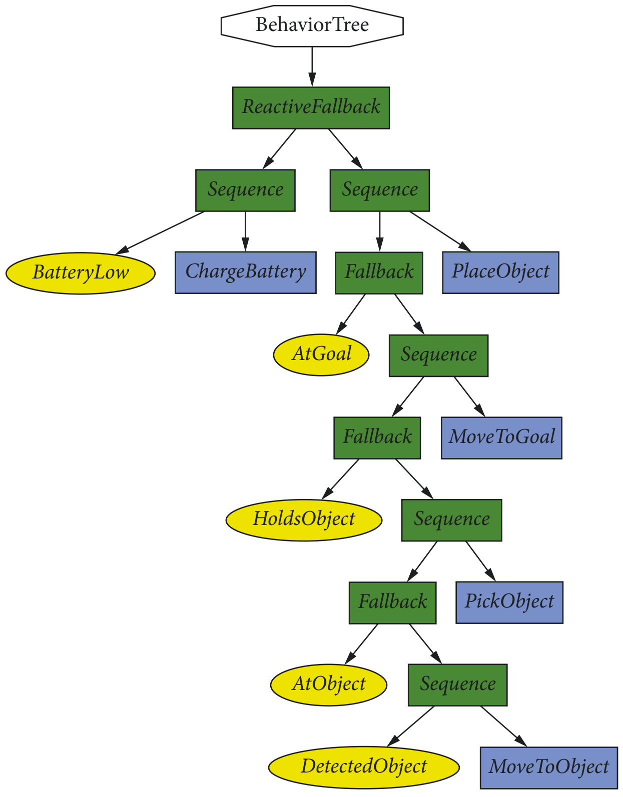


FIGURE 12: The reactive deliberative plan by adding a contingency on battery level.

The purpose of a skill is changing the state of the world what the robot lives in. By using its sensors, the robot can discover which objects are present in its world and where these objects are probably located and what their status is. Using sensor fusion and prediction, the robot can derive a WorldModel out of these observations, containing all beliefs that the robot has about its world. Skills can be implemented in a reusable manner by parameterization. Objects and locations can for instance be passed to skills as names of objects or locations that are attached to physical objects and locations by a lookup in the WorldModel.

For task planning and execution, it is important to note that a skill has a precondition to be successful and an effect, i.e., a prediction of the postcondition after its execution. For the use of automatic planners, skills with their precondition and effect are usually formally defined using, e.g., PDDL [19].

The executor (sequencer) is the software module that coordinates the execution of a task plan by sequencing skills. In our case, the executor (sequencer) is a BT engine, i.e., a software module that instantiates a BT in memory and ticks it. The structure of the BT may be hardcoded in the source code but may also be derived dynamically from a specification file (in XML or JSON format). We prefer the latter.

A common platform for robot software development is the Robot Operation System (ROS (https://www.ros.org/)) [26]. ROS is a set of software libraries and tools for building robot applications. ROS2 [27], the next generation of ROS, is well suited for industrial applications. A successful BT execution architecture should be able to fit in this infrastructure.

Production quality, opensource Behavior Tree libraries, and editors for robotic applications are available [7]. A good example is *BehaviorTree.CPP* (https://github.com/BehaviorTree). This open source library is currently being used in essential ROS2 packages, such as the Nav2 (https://navigation.ros.org) navigation stack [8]. It is a C++ based library that reads and runs behavior trees specified in XML. These XML files can be created using any text editor but also a nice graphical editor is provided, called Groot (https://github.com/BehaviorTree/Groot). This editor will not only be applicable to edit a Behavior Tree but does also monitor a BT-execution in real-time or enables replay of a trace of its execution.

When applying the BehaviorTree.CPP for your BT-implementation, all of the required BT control-, decorator-, condition- and action nodes need to be implemented in C++. A couple of frequently used BT nodes are provided. When other application or domain specific nodes are needed, they must be implemented as C++ plugins. Various base-classes are provided to make it easier to construct such new behavior tree plugins. A good practice is to make these plugins as thin as possible and make them delegate the work to specialized servers.

BT engines do usually support some kind of a blackboard. This way parameters can be set by one node and read by another. BehaviorTree.CPP action and condition nodes can exchange data via named and typed ports on a blackboard. All input and output ports, including their name and type, must be declared in the source code (header file) of a node.

In ROS, an application is a computation graph of interacting ROS nodes. ROS nodes interact by using any of the following interface concepts: topics, services, and actions. Topics are meant to be used for publish/subscribe communication of continuous data streams (e.g., sensor data, robot state). Services are meant for remote procedure calls that terminate quickly, e.g., for querying or setting the state of a node or for calling a quick calculation. Actions should be used for activating long running activities such as making a plan or moving a robot to goal pose. An action can be preempted, i.e., cancelled before completion (see Figure 13).

A BT sequencer that is implemented in ROS should be a node that runs the BT engine. It enables the BT engine to read the BT structure from an XML file and instantiates the nodes and it makes it possible to tick the root node of the BT with a fixed frequency, which will be somewhere between 10 and 1000 Hz.

The BT action and condition plugins would act as action clients of ROS Action Servers or service clients of ROS Service Servers. Action and Service Servers would be implemented in other nodes that could have been written in C++ but could be implemented in Python as well. This is exactly how the ROS2 Nav2 navigation stack works [8].

6. Summary

Behavior Trees (BTs) are very well suited to model and implement the robot behavior. Their modularity is a big advantage because it enables robot engineers to quickly develop, adapt, and extend the behavior of a robot. Furthermore, the visual representation of the BTs makes them

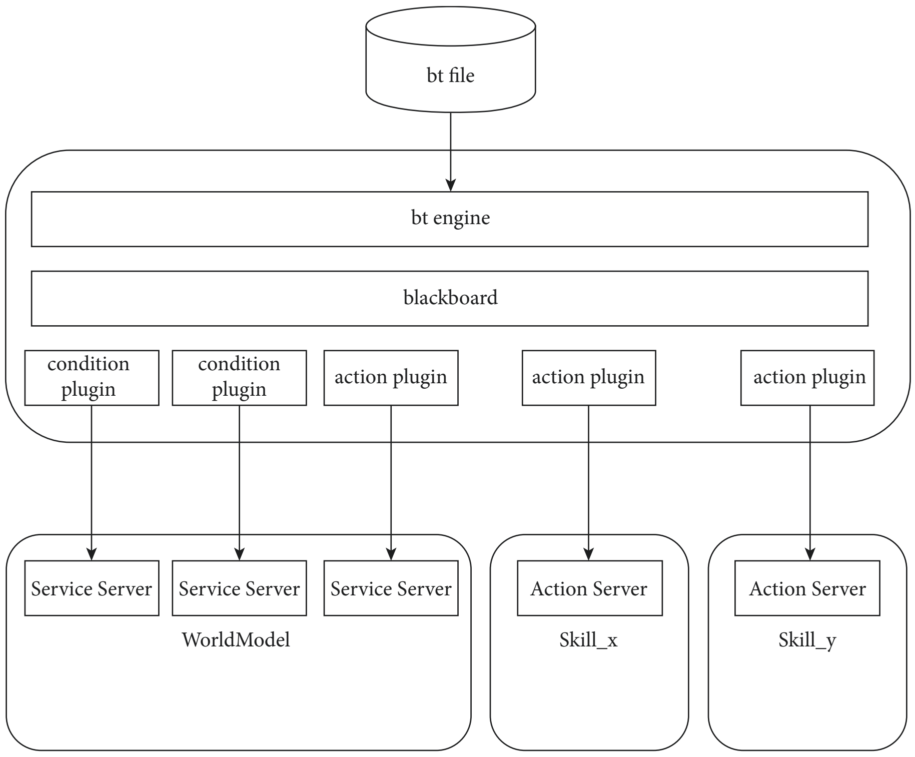


Figure 13: Behavior tree execution architecture.

easier to read and to be modified compared to (larger) hierarchical state machines (HSM); in particular, when supported by a graphical editor.

This paper provides a hands-on approach for robot software developers to apply BTs in their robot designs by: (1) a way of working for the stepwise construction of robust and reactive BTs and (2) the definition of a flexible architecture to execute BTs for ROS(2) based robots.

Data Availability

The data used to support the findings of the study can be obtained from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by programming autonomous robots in the applied research projects Raak MKB Close Encounters and Raak Pro DurableCase. The authors thank Simon Jansen and Heico Sandee of Smart Robotics for enabling the use of the example in Section 1.

References

[1] R. Siegwart, I. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, MIT press, Cambridge MA, USA, 2011.

[2] S. Bøgh, O. Nielsen, M. Pedersen, V. Krueger, and O. Madsen, “Does your robot have skills?” in *Proceedings of the 43rd International Symposium on Robotics*, Taipei, China, 2012.

[3] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI—an Introduction*, Chapman & Hall, London, UK, 2018.

[4] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, “Towards a unified behavior trees framework for robot control,” in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Philadelphia, PA, USA, 2014.

[5] M. Colledanchise and P. Ögren, “How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees,” *IEEE Transactions on Robotics*, vol. 33, no. 2, pp. 372–389, 2017.

[6] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, “A survey of Behavior Trees in robotics and AI,” *Robotics and Autonomous Systems*, vol. 154, Article ID 104096, 2022.

[7] R. Ghzouli, T. Berger, E. Johnsen, S. Dragule, and A. Wąsowski, “Behavior trees in action: a study of robotics applications,” in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language*, Chicago, IL, USA, 2020.

[8] S. Macenski, F. Martín, R. White, and J. Clavero, “The marathon 2: a navigation system,” IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Daejeon, Republic of Korea, 2020.

[9] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. Hager, “CoSTAR: instructing collaborative robots with behavior trees and vision,” in *Proceedings of the Robotics and Automation (ICRA) IEEE International Conference*, Philadelphia, PA, USA, 2017.

[10] M. Colledanchise and P. Ögren, “How behavior trees generalize the teleo-reactive paradigm and and-or-trees,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Las Vegas, NV, USA, 2016.

[11] P. Ögren, “Increasing modularity of UAV control systems using computer game behavior trees,” in *Proceedings of the AIAA Guidance, Navigation and Control Conference*, Minneapolis, MN, USA, 2012.

[12] F. Rovida, M. Crosby, D. Holz et al., “SkiROS—a skill-based robot control platform on top of ROS,” in *Robot Operating System (ROS)*, A. Koubaa, Ed., Springer, Berlin, China, 2017.

[13] Z. Cai, M. Li, W. Huang, and W. Yang, “BT expansion: a sound and complete algorithm for behavior planning of intelligent robots with behavior trees,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 7, pp. 6058–6065, Washington, DC, USA, 2021.

[14] M. Colledanchise, D. Almeida, and P. Ögren, “Towards blended reactive planning and acting using behavior trees,” in *Proceedings of the International Conference on Robotics and Automation (ICRA)*, Philadelphia, PA, USA, 2019.

[15] F. Martín, M. Morelli, H. Espinoza, F. Lera, and V. Matellán, “Optimized execution of PDDL plans using behavior trees,” in *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS ’21)*, Richland, SC, USA, 2021.

[16] P. Ögren, “Convergence analysis of hybrid control systems in the form of backward chained behavior trees,” *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 6073–6080, 2020.

[17] C. Paxton, N. Ratliff, C. Eppner, and D. Fox, “Representing robot task plans as robust logical-dynamical *systems*,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Macau, China, 2019.

[18] M. Winikoff and L. Padgham, “The Prometheus methodology,” in *Developing Intelligent Agent Systems: A Practical Guide to Design*, L. Padgham and M. Winikoff, Eds., John Wiley & Sons, Hoboken, NJ, USA, 2004.

[19] M. Ghallab, A. Howe, C. Knoblock et al., “PDDL—the planning domain definition language,” in *Proceedings of the AIPS-98 Planning Competition Committee*, Pittsburgh, PA, USA, 1998.

[20] E. Gat, “On three-layer architectures,” in *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. Bonnasso, and R. Murphy, Eds., MIT/AAAI Press, London, UK, 1998.

[21] R. Bonasso, D. Kortenkamp, D. Miller, and M. Slack, “Experiences with an architecture for intelligent, reactive agents,” *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 237–256, 1995.

[22] Y. Jiang, N. Walker, M. Kim et al., “LAAIR: a layered architecture for autonomous interactive robots,” in *Proceedings of the AAAI Fall Symposium on Reasoning and Learning in Real-World Systems for Long-Term Autonomy*, Arlington, VA, USA, 2018.

[23] R. Brooks, “A robust layered control system for a mobile robot,” *IEEE Journal on Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

[24] M. R. Pedersen, L. Nalpantidis, R. S. Andersen et al., “Robot skills for manufacturing: from concept to industrial deployment,” *Robotics and Computer-Integrated Manufacturing*, vol. 37, pp. 282–291, 2016.

[25] L. de Koning, J. Mendoza, M. Veloso, and R. van de Molengraft, “Skills, tactics and plays for distributed multi-robot control in adversarial environments,” in *RoboCup 2017: Robot World Cup XXI*, H. Akiyama H., Ed., Springer, Berlin, Germany, 2018.

[26] M. Quigley, B. Gerkey, K. Conley et al., “ROS: an open-source robot operating system,” in *Proceedings of the ICRA Workshop Open Source Software*, Philadelphia, PA, USA, 2009.

[27] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, 2022.