Automatic generation of behavior trees for the execution of robotic manipulation tasks

Parikshit Verma
Inst. of Industrial and Control Eng.
Universitat Politcnica de Catalunya
Barcelona, Spain
parikshit.verma@estudiantat.upc.edu

Mohammed Diab
Dept. of Electrical and Electronic Eng.
Imperial College London
London, UK
ORCID: 0000-0002-5743-5190

Jan Rosell
Inst. of Industrial and Control Eng.
Universitat Politcnica de Catalunya
Barcelona, Spain
ORCID: 0000-0003-4854-2370

***Abstract*—Robots should be able to exercise reasoning in both symbolic and geometric levels in order to plan a manipulation task. The execution of such tasks needs to be robust enough to cope with real environments. In an attempt to address this pertinent industry need, the paper proposes the use of behavior trees for effective robotic manipulation in dynamic environments. This paper presents a method to automatically generate a behavior tree and showcases its ability to enable the robot to reason at different levels and adapt to an uncertain and changing environment. This allows for a complex task to be robustly executed, pioneering the advancement towards fully functional service robots.**

***Index Terms*—Robotic manipulation, task and motion planning, execution manager.**

I. Introduction

With the latest paradigm of a robot being safe and conducive around humans and their ability to carry out complex tasks with maximum autonomy, there is a visible shift in the trend of robotics, from industrial robots to increasingly service robots. In this scenario, Task and Motion Planning (TAMP) plays an important role, since the automatic execution of any robotic manipulation tasks involves reasoning at both symbolic and geometric levels, as well as the ability to adapt/react to uncertain and changing environments. The reasoning at a symbolic level is generally considered as a task planning problem, whereas the reasoning at geometric level is considered as a motion planning problem. Planning is done off-line, although on-line adaptation or replanning may be necessary to avoid a failed task execution.

For instance, consider a case where a robot is employed as a bartender in a restaurant. After receiving the drink order from the customer, with symbolic information of the environment, the robot generates a task plan consisting of the sequence of robot actions to be performed and, for each robot action, a robot trajectory is generated using the geometric information of the environment. During the plan execution, any potential change in the environment (like the drink being unavailable in the storage or the drink storage location being changed or the pose of the drink being changed) may preempt an action to start due to the non-satisfaction of the action preconditions, or may lead to a failure in the action outcome. This gives rise to the need to replan the tasks for the new state of the environment or adapt/replan the trajectory of a specific action.

To cope with these issues, the problem addressed in this paper is, first, the off-line automatic generation of the code needed by a task manager to execute a complex task and motion plan and, secondly, the on-line adaptation of the sequence of tasks and motions to comply to the actual situations encountered while executing the task.

In this paper, the task planning is done using the Fast-Forward planner (FF, [1]), which does a heuristic search in state space to produce a symbolic-feasible sequence of robot actions that bring the initial symbolic state to the goal one. For each robot action in the plan, a geometric-feasible path is searched using The Kautham Project (TKP, [2]), a motion planning framework based on the Open Motion Planning Library suite of sampling-based motion planners (OMPL, [3]). The linkage between the symbolic actions and the geometric information is done through an interfacing layer defined as an XML configuration file, following the TAMP framework proposed in [4]. In the configuration file, the geometric description of each symbolic robot action is provided, e.g. for a Pick action there is information of the robot start configuration and of the goal (grasping) configuration (or alternatively, if the inverse kinematics of the robot is available, of the potential grasp poses where the gripper could be located w.r.t the object reference frame to pick the object). The TAMP framework is based on a Python client that calls the task planning service and the motion planning service offered by FF and The Kautham Project, respectively, using the Robotic Operation System (ROS, [5]).

The TAMP framework proposed in [4] is extended here with the capability to automatically write an output XML file that represents the Behavior Tree (BT, [6]) that may allow to execute the task with a real robot using a BT executor. Behavior Trees are an alternative to Finite State Machines (FSMs, [7]) as an execution manager for task execution in real-time. Since BTs can be represented in an XML format, to execute the TAMP problems in a real robot, the framework simply needs to generate the behavior tree XML file once the TAMP problem is solved and prior to the real execution. Moreover, the ability of BTs to be edited during run time and

This work was partially supported by the Spanish Government through the project PID2020-114819GB-I00

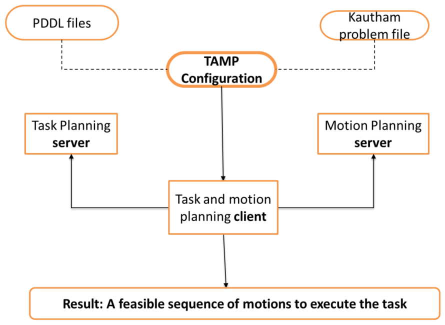
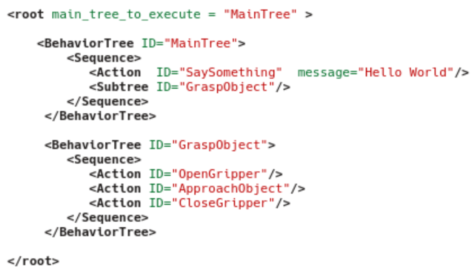


Fig. 1. TAMP framework.



```
<root main_tree_to_execute = "MainTree" >

    <BehaviorTree ID="MainTree">
        <Sequence>
           <Action  ID="SaySomething"  message="Hello World"/>
           <Subtree ID="GraspObject"/>
        </Sequence>
     </BehaviorTree>

     <BehaviorTree ID="GraspObject">
        <Sequence>
           <Action ID="OpenGripper"/>
           <Action ID="ApproachObject"/>
           <Action ID="CloseGripper"/>
        </Sequence>
     </BehaviorTree>

</root>
```

Fig. 2. XML Representation of Behavior Trees.

the fact that one can design reactive systems with BTs, makes BT executor a robust execution manager. The BT executor will have access to the actual robot controllers, both planing levels and the interfacing layer and all the communications will be based on ROS.

The paper is organized as follows. First, Sec. II and III review the TAMP framework and Behavior Trees, respectively, and then Sec. IV proposes the structure and automatic generation of behavior trees for TAMP purposes. Finally, Sec. V sketches the conclusions and future work.

II. The planning framework

This section summarizes the TAMP planning framework introduced in [4] and sketched in Fig. 1.

For planning the task at symbolic level, the FF [1] planner is used. FF is a domain independent planning system that can handle planning tasks specified in the Planning Domain Definition Language (PDDL, [8]). A ROS server has been implemented to wrap the FF planner. The FF service requires two PDDL files, one describing the domain and the other describing the problem. The domain file consists of actions, pre-conditions and post-conditions; the problem file has the information about the world, the initial state and the goal state. The service computes a feasible sequence of robot actions that will be called Action Plan.

For making a motion plan, TKP [2] is used. TKP is a software tool that provides different planners for motion planning, as well as visualization tools to view the trajectories. The main core of motion planners is provided by OMPL [3]. TKP has a ROS interface and can be accessed via ROS services. The motion planning problem is defined with an XML file (the Kautham problem file) with data regarding the poses and geometries of the obstacles and the robot, as well as the planner to be used, its parameters and the query to be solved. The geometric information can be accessed through symbolic labels defined in the Kautham problem file. The TAMP framework uses some of the TKP services to set the initial scenario and to set different queries to be solved.

The interfacing layer between the symbolic level and geometric level is implemented as an XML configuration file that contains the geometric information of each of the robot actions defined in the domain PDDL file. This file is used by the TAMP manager, which is a client to the task and motion planning services. The TAMP manager first calls the task planning service and then, for each robot action in the plan, calls the motion planning service, using the information provided by the configuration file.

III. Behavior trees

Behavior Trees can be seen as an alternative to Finite State Machines. The difference between the two is that the conditions and the states are coupled in FSM but in BTs the conditions are coupled with actions. A set of action-condition is generally called a behavior. Due to the special building blocks of BTs, BTs provide modularity and re-usability of these behaviors. BTs are also more easy to read and maintain.

A. BT nodes

The building blocks of BTs are known as BT-nodes. These nodes are broadly classified into two classes: *Control Nodes* and *Execution Nodes*. The control nodes help in regulating a periodic signal called *tick* which is generated by the root of the tree, whereas the execution nodes query the robot hardware to either get a feedback from environment or perform a robot action. The control nodes are mainly of four types:

- *Sequence Node*: A sequence node is a control flow node which regulates tick amongst its multiple child nodes, one child node at a time, in a given sequence (left to right manner). If any of the child nodes returns a failure, the sequence node returns a failure. In case all child nodes return a success, the sequence node returns success.
- *Parallel Node*: A parallel node is a control flow node which regulates tick amongst multiple child nodes simultaneously. If a user-defined number of the child nodes returns failure, the parallel node returns failure. In case all child nodes return success, the parallel node returns success.
- *Fallback Node*: A fallback node can be seen as a logical OR. This node regulates the tick amongst its multiple child nodes in a given sequence (left to right manner). As soon as one of the children returns success, the Fallback node returns success. In case all child nodes return failure, the fallback node returns failure.
- *Decorator Node*: These nodes are used to manipulate the output of a child node. For example, the *invert-the-output* decorator returns a failure when the child node returns a success, and vice-versa.

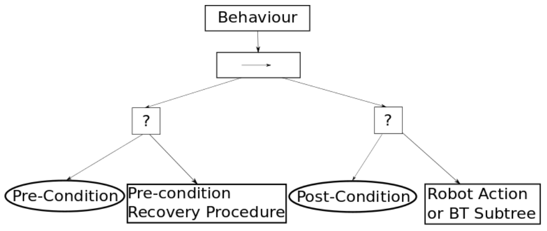


Fig. 3. General Behavior Representation.

In this work, sequence nodes, which are represented as an right arrow, and fallback nodes, which are represented with a question mark, will be used.

There are two types of execution nodes:

- *Action*: The action nodes do not have child nodes. These nodes can return Success, Failure or Running. Due to the Running state of the Action Node, these nodes can be preempted. These are represented as boxes in BTs.
- *Condition*: The condition nodes do not have child nodes. These nodes can return success and failure only. These are represented as circles in BTs.

B. XML description

Mathematically, BTs can be represented as directed acyclic graphs. Hence the BTs can easily be described in an XML file, as that shown in Fig. 2, where: a) the root defines the main behavior tree to be executed; b) each behavior tree has its own unique ID; c) the execution nodes are action nodes (no condition nodes are included in this example); d) with the Subtree tag one behavior tree can point to another one. Moreover, each node in a behavior tree can have input and output data ports which provides flexibility in exchange of data between nodes and between different behavior trees. This is done using a Blackboard which is a key/value storage shared by nodes of a tree. The input and output ports are connected by using the same key of the Blackboard.

C. ROS actions and services

To implement BTs for task and motion planning problems, the behaviorTree.ROS [9] library is used. This library gives an easy class implementation of BTs. The classes provide a way of initializing each BT node as a ROS node during the time of execution. Hence these BT nodes act as clients to ROS action servers and service servers. As the BT action nodes can be preempted, they are implemented as clients to ROS Action Servers and BT condition nodes are implemented as clients to ROS Service Servers. The exchange of information from a BT client to a ROS server is done via input-output ports of BTs.

IV. Proposed approach

This section proposes a BTs structure to execute the sequence of actions of a manipulation task generated by the TAMP framework described above, and how to automatically generate the XML files that describe them. The proposal seeks to achieve robustness in the task execution and, with this aim, behaviors trees at task level and at action level are

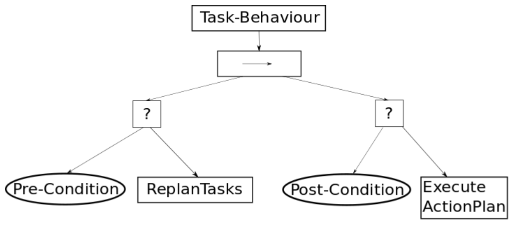


Fig. 4. Task Level behavior.

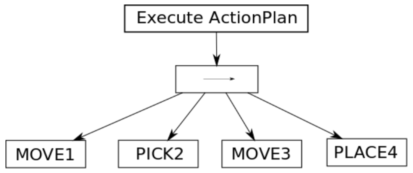


Fig. 5. *Execute ActionPlan* BT node for a given task.

designed with the general sequence structure shown in Fig. 3, where:

- First, the pre-condition for the behavior to be executed is checked and, upon failure, a recovery procedure is run so as to make the pre-condition be satisfied.
- Second, the post-condition of the behavior is checked and, upon failure, the behavior is executed (by a BT action or a BT sub-tree) so as to make the post-condition be satisfied.

In the following subsections, the task level behaviors, action level behaviors and the automated generation are explained.

A. Task Level behavior: The main BT

A root BT with a fixed structure is defined to manage the execution at task level. This will be the main BT to be executed and contains a sub-tree with a fixed ID called *Task Level behavior* (see Fig. 4) that has:

- A pre-condition to check if the state of the environment detected by the perception system corresponds to the initial state used to obtain the Action Plan.
- A BT-action called *ReplanTasks* that restarts the planning at task and motion levels.
- A post-condition to check if the state of the environment detected by the perception system corresponds to the goal one as specified by the user.
- A BT-subtree called *Execute ActionPlan* to execute the Action Plan.

The *ReplanTasks* BT-action first updates the initial state with the current perceived one (by calling ROS services to make modifications in the PDDL problem file, the XML Kautham problem file and to the TAMP configuration file, regarding the symbolic and geometric information of the object locations). Then it calls the FF task planner service to obtain a new Action Plan and the Kautham motion planning service to find the motion paths for the actions.

The *Execute ActionPlan* is a sub-tree indicating the sequence of indexed robot actions given by the FF task planning service, assuming an initial state of the environment. The

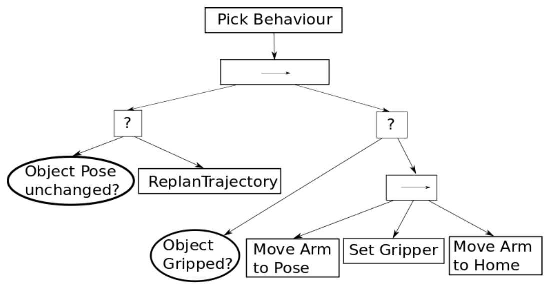


Fig. 6. Pick Action Level behavior.

robot actions are indexed, as illustrated in Fig. 5, since an action may need to be performed multiple times in order to achieve the final state, and there may be the need to replan the motion of a given particular action. Each indexed robot action points to another sub-tree which is called *Action Level behavior* explained next.

B. Action Level behaviors: BTs for task actions

Like the Task Level behaviors, the Action Level Behaviors have a fixed format, following the general BT structure described in Fig. 3, and that will be particularized for each type of robot action. As the BTs can be edited at run time, the instances of Action Level Behaviors can be updated when the initial state changes and the action plan is updated. As an example, Fig. 6 shows the Pick Action Level Behavior where:

- The pre-conditon checks if the perceived pose of the object to be grasped is the same as the nominal object pose that was used to compute the motions.
- The recovery action to be done (upon failure of the pre-condition) consist in, first, calling the service to update the geometric information of the object and, then, in recomputing a collision-free grasping motion using the Kautham motion planning service.
- The post-condition checks if the gripper of the robot is full, meaning that the pick operation has been successful.
- The Pick action consists in a sequence of BT-actions, each one performed by the corresponding ROS service action of the robot and gripper: a) move arm to grasp configuration; b) close gripper; c) move arm to home configuration.

The *ReplanMotion* BT-action first updates the geometric information of the object to be picked and then queries TKP services to obtain a collision free trajectory.

C. Automatic generation and execution of XML files

Following the BT structure proposed, the TAMP framework described in Sec. II has been extended so as to output the XML files of the BTs. Now the TAMP manager starts by writing the XML of the Task Level Behavior, which is independent of the problem to be solved. Then, after calling the task planning service, the manager writes the XML file for the *Execute ActionPlan* BT, as in Fig. 5, hence completing the first instance of Task Level Behavior.

Then the TAMP manager manages each of the actions in the plan using the information of the TAMP configuration file and calling the motion planning service when necessary. Moreover, the TAMP manager also writes the XML file corresponding to the Action Level Behavior BT of the action. This includes the coding of the trajectory generated by the motion plan as required by the robot ROS action service. Once all the actions have been managed, the first instance of all the Action Level Behaviors is complete.

The generated BT XML files (the Task Level Behavior and the Action Level Behaviors) are passed to the BT executor which is responsible for initialising the tree, ticking the nodes of the BT and monitoring the state of the Task level and Action level behaviours. If a change in state is observed, the tree is re-initialised, hence executing the task and motion planning problem in a robust manner.

V. Conclusions and future work

This paper has proposed a procedure to generate behavior trees (BTs) which are able to execute a robot manipulation task in possibly uncertain and changing environments. The approach is integrated within a task and motion planning framework that computes the sequence of actions and associated motions to perform the manipulation task, and that automatically writes the XML files that describe the BTs. The proposal takes advantage of the ability of BTs to be edited during run time, allowing adaptation of the action plan or of the trajectories to changes in the state of the environment.

Intelligent perception systems and reasoning mechanisms will be integrated in the future to allow the robot to interpret failures and provide recovery strategies. Also, testing this approach in the real scenarios with the TIAGo robot will be considered. Starting with just Move, Pick and Place actions, manipulation tasks will be designed to show how the robustness of the proposed framework may enable service robots to be easily used in both industrial and domestic environments.

References

[1] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, pp. 253–302, 2001.

[2] J. Rosell, A. Pérez, A. Aliakbar, Muhayyuddin, L. Palomo, and N. García, "The Kautham Project: A teaching and research tool for robot motion planning," in *IEEE Int. Conf. on Emerging Technologies and Factory Automation*, 2014.

[3] I. Sucan, M. Moll, L. E. Kavraki *et al.*, "The open motion planning library," *Robotics & Automation Magazine, IEEE*, vol. 19, no. 4, pp. 72–82, 2012.

[4] S. Saoji and J. Rosell, "Flexibly configuring task and motion planning problems for mobile manipulators*," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 1285–1288.

[5] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, vol. 3, 2009, p. 5.

[6] M. Colledanchise and P. gren, "Behavior trees in robotics and AI," Jul 2018. [Online]. Available: http://dx.doi.org/10.1201/9780429489105

[7] G. O'Regan, *Automata Theory*, 2016, p. 117126.

[8] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL—The Planning Domain Definition Language," 1998.

[9] D. Faconti, "Behavior Tree Ros Library," https://github.com/BehaviorTree/BehaviorTree.ROS.