2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
September 24–28, 2017, Vancouver, BC, Canada

Extended Behavior Trees for Quick Definition of Flexible Robotic Tasks

Francesco Rovida[1], Bjarne Grossmann[1] and Volker Krüger[1]

Abstract— **The requirement of flexibility in the modern industries demands robots that can be efficiently and quickly adapted to different tasks. A way to achieve such a flexible programming paradigm is to instruct robots with task goals and leave planning algorithms to deduct the correct sequence of actions to use in the specific context. A common approach is to connect the skills that realize a semantically defined operation in the planning domain - such as picking or placing an object - to specific executable functions. As a result the skills are treated as independent components, which results into suboptimal execution. In this paper we present an approach where the execution procedures and the planning domain are specified at the same time using solely extended Behavior Trees (eBT), a model formalized and discussed in this paper. At run-time, the robot can use the more abstract skills to plan a sequence using a PDDL planner, expand the sequence into a hierarchical tree, and re-organize it to optimize the time of execution and the use of resources. The optimization is demonstrated on a kitting operation in both simulation and lab environment, showing up to 20% save in the final execution time.**

Keywords: autonomous robots, planning, behavior trees, hierarchical task networks, skills

I. INTRODUCTION

Modern robotics in factory environments is characterized by a need for increased flexibility. The flexible manufacturing paradigm implies a frequent adaptation of robot tasks, but current techniques require a large amount of time and programming effort to specify it. One proposal for simplifying the robot task programming defines tasks as sequences of skills, where skills are identified as the re-occurring actions that are needed to execute standard operating procedures in a factory (e.g., operations like pick object or place at location) [1]. The robot provides its services to the human operator at a level of goals (”what has to be done”) and uses automated planning techniques to figure out by itself how to fulfill the task goals with the available skill set. In this way, it can be re-configured quickly and becomes more adaptable to the context. The goal of increasing robot autonomy in the factory environment relies on two requirements: (i) the robot being able to receive goals and automatically sequence skills to fulfill them and (ii) the robot being able to manipulate that planned sequence for an optimal execution. The first requires the definition of an human-robot interaction space and jointly a domain for task planning: for this we use a supporting semantic world model that gets automatically translated, with a self-developed parser, into the Planning Domain Definition Language (PDDL), a syntax understandable from several classical planners. This implementation is available in our Skill-based platform for ROS (SkiROS), already presented in previous work [2]. To explain the second point we start from an example: consider the robot being assigned the task of collecting several objects at different locations. The planner would generate a sequence of pick and place skill, and each step within each skill would be executed, even if it is redundant. In addition, considering each skill separately, it would be difficult to have a smooth and optimal movement between the picking and placing locations. Finally, the skills would be logically executed sequentially, meanwhile some sub-operations could theoretically be done in parallel (e.g. locate next object meanwhile placing the current one). Optimizing the final execution requires a good integration of the planning and the executive layer, something generally missing in the existing solutions. A great extent relies on Hierarchical Finite-State Machines (HFSM) or Behavior Trees (BT) to script low-level procedures. Some famous robot architectures have been using Domain Specific Languages (DSLs), like TDL [3] and PLEXIL [4] for the CLARAty architecture[5], or BIP for the LAAS architecture[6]. All these solutions integrate with a deliberative layer only to some extent, as the major part of the scripted procedure remains hidden to the planner. Only recently, researchers have been investigating more coherent solutions for the integration of plans and scripts [7], [8]. On the other side, it is theoretically possible to plan from the task all the way down to the primitive level, using partial order planners or advanced PDDL planners to parallelize execution. Nevertheless this is not considered as a suitable solution, because it is not scalable to a large number of primitives and, more important, doesn’t allow to integrate predefined sequences - defined by developers or learned - in the planning domain.

In this paper, we propose a novel formalism to design a robot behavior, the *extended Behavior Tree (eBT)*, an extension of the BT model, meant to merge coherently scripted and planned procedures into an unified representation. An eBT describes at the same time how to execute a *procedure* and its effects on the world state. Starting from atomic procedures, *primitives*, the user can define arbitrarily complex eBTs, *skills*, associated to actions in a planning domain. Given a goal state, a classical planner [1] is used to plan in the state space and initialize the robot eBT. Starting from the planned sequence, the tree is expanded by replacing the abstract nodes with skills and primitives and finally re-organized, depending on the context, using a modified HTN

[1] The authors are with the Robotics, Vision, and Machine Intelligence (RVMI) Lab at Aalborg University Copenhagen, 2450 Copenhagen, Denmark e-mail: francesco@m-tech.aau.dk, bjarne@m-tech.aau.dk, vok@m-tech.aau.dk

[1]For the tests we used the Fast Downward planner: http://www.fast-downward.org/

planner. The resulting plan is a tree comparable to a BT, directly executable and optimized, in our case, to minimize the execution time and the use of resources.

The paper contribution is the formal definition of the eBT model. Moreover, an algorithm to optimize the tree is presented and demonstrated on a real application example. Finally, a python implementation is provided open source on the web to allow the reproducibility of results.

The rest of this paper is organised as follows. First, the related work is considered, with a special focus on the formal model of BTs and HTNs. Then, the eBTs model is presented and the architecture integrating the model is introduced. The paper concludes with a set of experiments performed on a real example use-case that demonstrates the system functioning in a mock up factory environment.

II. Related work

During the last three decades, three main approaches to robot control have dominated the research community: reactive, deliberative, and hybrid control [9]. Most modern autonomous robots follow a hybrid approach [10], [5], [6], with a reactive layer, responding to immediate changes in the world [11], [12], managed from a deliberative layer, employing a sense-plan-act paradigm to work in a goal-directed manner [13]. In this context, the researchers are still focused on finding appropriate interfaces to link the declarative descriptions needed for high-level reasoning and the procedural ones needed for low-level control [3], [4], [14], [15].

This middle layer is commonly refereed to as executive layer, usually concerned in decomposing high-level tasks into low-level behaviors, monitoring execution and handling exceptions. A current common way to model procedures, is to use concurrent *hierarchical finite state machines (HFSMs)* [16], [17], [18]. HFSMs are used, for example, at WillowGarage on the PR2 robot as task coordinator for scenarios like playing pool, cleaning up the table with a cart and fetching beer from the refrigerator. HFSMs are easy to design, implement and interpret. The hierarchical organization reduces, to some extent, the explosion of states and events when the complexity of the HFSM increase. The main drawback of HFSMs are that the transitions between the states must be specified manually and remain fixed, so that the behavior of each scenario has almost no reuse. In fact the static composition of behaviors results in little capabilities for a situation dependent task execution or optimization.

A more recent approach, even though already used for more than a decade in AI game development to program the behavior of Non-Player Characters, are the *Behavior Trees (BT)* [19], [20]. They can be seen as an evolution of HFSMs by replacing states with procedural actions or tasks whereas the next state is given implicitly by the structure of the tree. They allow a much more natural way of defining behavior and remove the need of wiring different states manually into the code, hence promoting the re-usability of actions/nodes. Using BTs it is possible to construct and restructure program execution at run-time rather than at compile time. However,

Node type	Symb.	Execution
Root	$\varnothing$	return tick(child(0))
Sequence	$\rightarrow$	tick ch. sequentially. stop with F if one ch. F
Parallel	$\|\|$	tick ch. in parallel. stop with F if one ch. F
Selector	?	tick ch. sequentially. stop with S if one ch. S
Decorator	δname	varies
Action	$\square$	return S, F or R
Condition	$\circ$	if condition=True return S else return F

TABLE I
POSSIBLE NODE TYPES OF A BT [19]. (CH.: CHILD/CHILDREN)

BTs still doesn't provide any detail regarding the purpose of behaviors, resulting in no possibility of optimization.

The *Hierarchical Task Networks (HTN)* ([21] Chap. 11) is a planning method that tries to encapsulate the procedural description directly into the planning domain. HTN planning is like classical planning in that each state of the world is represented by a set of atoms, and each action corresponds to a deterministic state transition. However, the input for HTN includes, in addition to a set of operators similar to those of classical planning, also a set of methods, describing how to decompose some task into some set of subtasks (smaller tasks). This provides a convenient way to write problem-solving ”recipes” that correspond to how a human domain expert might think about solving a planning problem. HTN is at the moment one of the most widely used planning methods in practical applications [22], [6], [23]. For kitting operations classical planning has been applied by some researchers [24], [25], but without concerns regarding programming intuitiveness or optimization of execution time, as addressed in this paper.

III. Preliminaries

We extended the discussion of related work with a brief introduction of the terminology and the formal model of BTs and HTNs, in order to give to the reader the base to compare them with the eBT model presented later in the paper. The definitions are based on the work of Marzianotto et Al. [19] (for BTs) and Ghallab et al. Chap. 11 [21] (for HTNs).

A. Behavior Tree

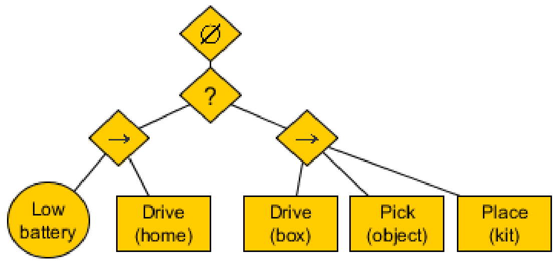

Fig. 1. A BT representation of a drive-pick-place sequence.

BT is a formalism for representation and execution of procedures. Following [19], we define a BT as a directed acyclic graph $G(V, E)$ with $|V|$ nodes and $|E|$ edges, using

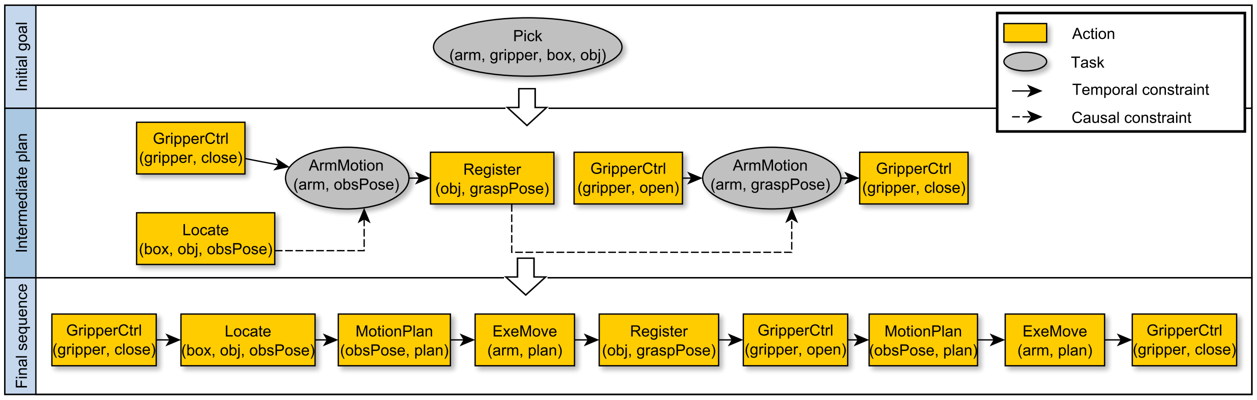
Initial goal
Intermediate plan
Final sequence
Pick (arm, gripper, box, obj)
Action
Task
Temporal constraint
Causal constraint
GripperCtrl (gripper, close)
ArmMotion (arm, obsPose)
Register (obj, graspPose)
GripperCtrl (gripper, open)
ArmMotion (arm, graspPose)
GripperCtrl (gripper, close)
Locate (box, obj, obsPose)
GripperCtrl (gripper, close)
Locate (box, obj, obsPose)
MotionPlan (obsPose, plan)
ExeMove (arm, plan)
Register (obj, graspPose)
GripperCtrl (gripper, open)
MotionPlan (obsPose, plan)
ExeMove (arm, plan)
GripperCtrl (gripper, close)

Fig. 2. Expansion of an HTN. Dashed lines represent the link created by pre and post constraints. The pick task (top) is expanded with an intermediate method. Secondly, the ArmMotion tasks are expanded with another method, leading to a final ordered sequence (bottom).

the usual definition of parents and children for neighboring nodes. The node without parents is called the root node, and nodes without children are called leaf nodes. Each node in a BT, with the exception of the Root, is one of six possible types, summarized in Table I. The Root periodically, with frequency f_{tick}, generates an enabling signal called tick, which is propagated through the branches according to the algorithm defined for each node type. When the tick reaches a leaf node, it executes one cycle of the Action or Condition and returns the node state value. Actions can alter the system configuration, returning one of three possible state values: Success(S), Failure(F), or Running(R). Conditions cannot alter the system configuration, returning one of two possible state values: Success, or Failure. The basic processor types are sequence, parallel and selector. Decorators are used to extend the basic types in order to create new types of execution. Typical example are the δloop decorator, that continue to tick the children until one fails, or the δforceSuccess decorator, that returns S, no matter the return value of its children. The literature does not set precise guidelines for the decorators types, that are left open to user creativity. An example of a BT is presented in Fig. 1. In this case, if the robot battery is low, the robot will move back to home position. Otherwise, it will execute a sequence of drive, pick and place. Note that in order to pass information between nodes, a common approach is to use a set of shared variables names, called *blackboard*. Between the brackets of each action are defined the variables, targeting the respective value in the shared blackboard.

B. Hierarchical Task Network

HTN is a formalism for planning sequences of actions. Like classical planning, each state of the world is represented by a set of atoms, where an atom p is considered true in state s if $p \in s$ or false otherwise. Each action corresponds to a deterministic state transition, adding or removing atoms from the state on which it is applied. An action a is defined as a tuple $(name, precond, effects)$, where $name$ is a unique action symbol, $precond$ is a set of literals (positive and negative atoms) that must be valid before applying the action and $effects$ is a set of literals that are added or removed from the state when the action is applied. A task t is an abstract operation, that during the planning procedure is either replaced by an action with $name = t$ and applicable in the state s, or it is expanded using a *method*, that describes how to decompose the task into some set of subtasks (smaller tasks). A *method* is defined as a 4-tuple:

$$m = (name, task, subtasks, constr)$$

where:

- *name* - is an expression of the form $n(x_0 \dots x_k)$, where n is a unique method symbol and $x_0 \dots x_k$ are the parameters
- *task* - identifies the task on which the method can be applied
- *(subtasks, constr)* - is a task network that will replace the task

A *task network* is defined as a pair $w = (subtasks, constr)$, where subtasks is a set of task nodes and constr is a set of constraints between the nodes. The constraint types are:

- precedence constraint - constraining the order of two nodes
- pre, post and between constraints - requiring a state variable to be true respectively just before a node execution, just after a node execution and between two nodes executions

The HTN planning problem is defined as a 4-tuple $P = (s_i, w, O, M)$, where s_i is the initial state, w is the initial task network, O is a set of actions, and M is a set of methods. Finding a solution means finding a total ordered sequence of actions, with all parameters grounded, executable in the state s_i and respecting all defined constraints. The planning algorithm starts with a root task that represents the problem domain and proceeds by decomposing the tasks recursively into smaller and smaller subtasks, until only actions are left. In Fig. 2 we present an example of the planning procedure, where the pick task (top) is expanded in two steps. Note that the final sequence is just one of the possible total ordered sequences (e.g. by definition locate and GripperCtrl could be swapped). The final ordering depends on the heuristic embedded in the planner.

IV. Extended Behavior Trees

In [19] we read: "the flexibility and modularity of BTs, make them a good candidate for representing Middle Layer plans that are created and maintained automatically by high-level AI algorithms". Nevertheless, we argue that by themselves BT presents a structure that does not fit completely when used in conjunctions with planners. For example, conditions, considered as nodes in BT, do not make sense for a planning domain where they are intrinsically present as part of every node. Moreover, when we consider task planning, the final output is usually not a tree, but rather a sequence of high-level actions. To reach the goal of a coherent integration of planned and predefined procedures, a new representation is necessary. In this section we present and formalize the core contribution of this paper, the Extended Behavior Trees (eBTs) model, that joins BT and HTN paradigms into a single one. Until nowadays the two models have been generally considered disjointly, but our contention is that the BT model, intuitive to use and define, can be modified to incorporate information necessary to define an HTN domain and therefore extend the BT flexibility with HTN planning. Since by themselves HTN planning is not meant to build up plans from goals, we suggest to use a classical planner based on PDDL representation for initializing the HTN task decomposition and also for re-planning in case of failures during execution.

Model	Hier. decomposition	Scripting	Planning	Mixed script-plan	Abstract procedures	Parallel exe.
Hierarchical Finite State Machine	✓	✓	✗	✗	✗	✓
Behavior Tree	✓	✓	✗	✗	✗	✓
Hierarchical Task Network	✓	✗	✓	✗	✓	-
STRIPS	✗	✗	✓	✗	✗	-
Ext. Behavior Tree	✓	✓	✓	✓	✓	✓

TABLE II
Characteristics of different models: present (✓), not present (✗) or not evaluable (-).

A. Concept and motivation

In Table II we identify some desirable behavior model's characteristics that we manage to pull together with our proposal. *Hierarchical decomposition* is desirable to keep the behavior complexity localized and manageable. Both BT and HTN divide procedures hierarchically. Nevertheless, meanwhile BT has a tree structure, the HTN's hierarchy is resolved during planning. In HTN operators replace abstract tasks so that the final output is a sequence (Fig. 2). eBT keep the hierarchical structure during expansion, leading to a final output that is a tree, executable similarly to a BT.

The possibility of *scripting* some behaviors is desirable, in particular in robotics, since some procedures are easier to describe logically than declaratively. Others are so intrinsically coupled, that do not justify to be considered in the planning problem.

Automated *planning* is desirable to keep the behavior flexible to the context. In BT we miss the concept of pre- and postconditions, that is necessary to define the purpose of a node and therefore to evaluate why and when to use it. Conditions are present as nodes and are used to execute conditionally sub-branches of the tree. In HTN the nodes embed a list of conditions, used to plan the final ordered sequence. eBT, like HTN, embeds conditions in the nodes and doesn't have condition nodes.

Abstract procedures are necessary for real modularity. eBT allows procedural abstraction, meaning that we can have different implementations with the same input and semantical meaning, e.g. we could have an implementation to pick from boxes and another one to pick from a tabletop. Additional preconditions allow the planner to choose which implementation to instantiate at run-time.

In BT the children's sequence in a node is totally ordered and can't be modified at runtime. In HTN, only the necessary precedence constraints are defined and the other nodes are left unordered. eBT specifies a total order between nodes: nodes can't be left unconnected. Nevertheless, the order is considered as a *suggestion* from the planner, that is allowed and expected to reorganize the sequence in order to optimize, in our case, execution time and use of resources.

B. World model

We collect the necessary global information into a semantic world model [26] that is shared among all eBT's nodes. The world model is divided into two parts: a static *ontology*, describing a taxonomy of classes and relations that can appear in the scene, and a dynamic *scene*, a graph where nodes are objects and edges are semantic relations between them. An *object* is defined as a tuple with: *id*, *type*, *label* and *properties*. *Type* and *label* map the object to a class and an individual template in the ontology, *id* is a unique integer identifier of the object in the scene (abstract objects have id = -1, instantiated objects have id>=0) and *properties* is a list of tuples $(key, value)$, where key is a string, and $value$ is some data associated to the object (e.g. position and orientation). Objects are symbols of human intuitive concepts that can be both concrete (e.g. box, gripper) or abstract (e.g. grasping pose, trajectory). They are are used to parametrize the eBTs and generate the planning domain.

C. Formalization

We define an eBT as a directed acyclic graph $G(V, E)$, like defined in section III-A, with few modifications. In eBT each node can be either an abstract *procedure (P)*, a *skill (S)* or a *primitive (T)*. Both P, S and T are tuples, defined as:

$$P =< type, params, conds(params) > \quad (1)$$

$$S =< P, name, subconds(P.params), processor, children > \quad (2)$$

$$T =< P, name, subconds(P.params), processor > \quad (3)$$

where the terms are:

- **type** - a unique symbol identifying the abstract procedure.
- **name** - a unique symbol identifying the procedure instance.
- **params** - a set of parameters $x_1, \ldots, x_n$ that ground in world model's objects.
- **conds** - are pre-conditions and post-conditions, that must be valid, respectively, just before and just after the procedure execution. We assume 5 possible conditions:
  - isType(x_k, t): type(x_k) = t
  - isGrounded(x_k): id(x_k) >= 0
  - hasProperty(x_k, p): ∃ property(x_k)[p]
  - property(= property(x_k)[p] value): property(x_k)[p] = value
  - relation(x_k, predicate, x_j): ∃ [x_k, predicate, x_j]
- **subconds** - additional conditions that are added to the ones of the abstract procedure.
- **processor** - define the algorithm executed when the node is ticked. The algorithms are the same ones defined for BTs in Table I, except for the *condition* and the *selector* types, that are excluded in the eBT model.
- **children** - an eBT.

A tree is *expanded* when all the P nodes are replaced by S or T nodes. In words, a tree with at least an abstract procedure is *not-expanded*, otherwise it is *expanded*. When a tree is expanded, it can be executed.

D. Modify execution order

The planning algorithm consist in decomposing the initial procedure tree until all the leaf nodes are instantiated and are primitives. During the decomposition, the proposed algorithm rearrange the tree in order to optimize execution time and use of resources. For this, the new nodes are moved in front in the existing tree as long as the pre-conditions holds and independent nodes are organized for parallel execution. Changing the nodes order is in general not valid, as it can break the correct execution sequence. We therefore analyze the fundamental rules keeping the approach correct. We start by considering a flat procedure tree (a sequence), with only primitive nodes. Let ${}^{C}T^{k}_{E}(x)$ be a primitive k with preconditions C, postconditions E and parameters x. We define the application * of a primitive to a state s_0 as:

$$s_k = s_0 *^{C} T^{k}_{E}(x) = s_0 \cup E^{+}_{k}(x) - E^{-}_{k}(x) \tag{4}$$

Meaning that the positive conditions $E^{+}_{k}(x)$ are added and the negative postconditions $E^{-}_{k}(x)$ are removed from the state. This is valid iff all preconditions C are satisfied at state s_0, otherwise the application is undefined. Considering now the application of two primitives (here C, E and x are omitted for simplicity):

$$s_{ki} = s_0 * T^k * T^i = s_0 * \{T^k, T^i\} \tag{5}$$

Lemma 1: The commutative property of application * is valid iff:

1) s_0 satisfies the preconditions of T^i (no dependence)
2) s_i, the state after application of T^i, satisfies T^k preconditions (no interference)
3) $E^{+}_{k} \cap E^{-}_{i} = \varnothing$ and $E^{+}_{i} \cap E^{-}_{k} = \varnothing$. (not inconsistent)

Proof: If rule 1 is respected, then it is possible to apply T^i to state s_0:

$$s_i = s_0 * T^i \tag{6}$$

If rule 2 is respected, then it is possible to apply T^k to s_i, the state after application of T^i:

$$s_{ik} = s_i * T^k \tag{7}$$

Finally, if rule 3 is respected, we have:

$$s_{ki} = s_0 \cup E^{+}_{k} - E^{-}_{k} \cup E^{+}_{i} - E^{-}_{i} = s_0 \cup E^{+}_{i} - E^{-}_{i} \cup E^{+}_{k} - E^{-}_{k} = s_{ik} \tag{8}$$

■

Corollary 1: From Lemma 1, derives that we can apply in parallel a set of primitives S, if commutativity is valid for each pair (u, v) with $u, v \in S$.

We now expand to discussion to general trees. A skill with n children procedures is represented sequentially as:

$${}^{C}S^{s}(x)\ {}^{C}P^{1}_{E}(x_1)\ {}^{C}P^{\cdots}_{E}(x_{\ldots})\ {}^{C}P^{n}_{E}(x_n)\ S^{e}_{E}(x) \tag{9}$$

Where S^s and S^e denote respectively the start and the end of the skill and P are the children procedures. The preconditions C must be valid before the skill's start node, meanwhile postconditions E are applied after the skill's end node. By definition, S^e must always remain placed after the head and all the children. This is equivalent to say that the preconditions of S^e are the conjunction of children effects. Otherwise, lemma 1 is still valid for skill nodes and other procedures can move into (become child) or go out of the skill.

Lemma 2: a child P^i that goes out of the parent skill s, must inherit all the skill's preconditions that are applied over its parameters. Formally: $C_i(x_i) \leftarrow C_i(x_i) \cup C_s(x_{si})$, where $x_{si} = x_s \cap x_i$

To make an example: consider a module *arm_motion*, that moves the robot end-effector from an initial location to a final one. The module has as parameters the arm and the gripper. The arm_motion is child of a place skill P, that has a precondition C1, besides others, that the gripper should hold an object. If the *arm_motion* gets out of P^s, C1 must be still verified, otherwise the arm could be moved before any object is grasped.

An eBT skill can have two types of processors: sequential and parallel. The following rule applies:

- A procedure P moving into a skill S with parallel processor, must respect the application commutativity w.r.t. all the skill's children, following Corollary 1.

E. Plan and optimization algorithm

To access an eBT we employ a traversal structure, namely the *visitor pattern* [27]. This gives us the advantage of having various implementations like different traversal algorithms, simulated execution and planning strategies, without modifying the object structure.

The optimization is initialized by invoking the function $process(s, r, wm, im)$ 1, where s is an empty procedure, r is the eBT root node, wm is the world model (tracking the world state) and im manage the set of skills and primitives. s contains methods (*addStart, addEnd*) to add/remove a node to the sequence and simultaneously apply/remove the postconditions effects on the world state. *undo* allows to revert the execution and keeps track of the removed nodes, so that they can be reinserted with the *redo* command. The initial tree consist of a root node (with sequential processor) having as children the output sequence of the PDDL planner. The *process* function first invokes the *preProcessNode* 2, goes recursively into the node's children with a depth-first policy, and finally invokes the *postProcessNode* 3. First thing, *preProcessNode* tries to replace an abstract node with an implementation. If the postConditions are already satisfied, the node is removed. Otherwise, the node is pulled backwards in the sequence, as long as the rules presented in the previous section are respected. If the node stops, it is inserted with sequential processor: if the parent has not a sequential processor, *addStart* creates a new sequential node p and insert the current node c and the previous node as children of p. Then, if c has children the sequence is restored. Otherwise, the algorithm will jump to *postProcessNode* where c-end is added, possibly in parallel with subsequent nodes. This means that if (i) there is a node a to redo (meaning that c passed in front of another one), (ii) c and a have the *same* parent and (iii) the parent has not a parallel processor, a new parallel node is added having c and a as children.

On success, in s is stored an optimized procedure, equivalent to the input procedure.

Algorithm 1: process(s, node, wm, im)

```
1 if not preProcessNode(s, node, wm, im) then
2     return False
3 for c in children(node) do
4     if not process(s, c, wm, bb, im) then
5         return False
6 if not postProcessNode(s, node, wm, im) then
7     return False
8 return True
```

V. Evaluation

Model and algorithms presented in the paper have been implemented in a open source python library [2]. It allows to test the optimization algorithm presented in this paper and to see in detail the skill and primitive set defined for the experiments.

A. Setup

The kitting task, common in a variety of manufacturing processes, involves the navigation of a mobile platform to various containers from which the objects are to be picked and placed in their corresponding compartments into a kitting box, carried by the robot. We use the skills and modules developed for the factory kitting use-case, described in detail in Rovida et al. [2]. The skill set consists of three skills: *drive*, *pick* and *place*. These consist of the following combination of primitives (Fig. 4 (up)):

- **drive(Robot, Initial, Target)** - drives the robot to a container. Post:(Robot, at, Target)
- **locate(Camera, Container, Object)** - roughly localize an object on a flat surface using a camera. Pre: (Robot, at, Container) Post: isGrounded(Object), hasProperty(Object, Position)
- **locate_kit(Camera, Kit)** - localize a kitting-box using a camera. Pre: (Camera, hasViewOn, Kit) Post: hasProperty(Kit, Position)
- **registration(Pose, Gripper, Camera, Object)** - accurately localize an object using the arm camera. Pre: hasProperty(Object, Position), (Gripper, at, Pose), (Object, hasA, Pose) Post: hasProperty(Object, RegisteredPose)
- **move_arm(Arm, Gripper, Initial, Target)** - move the end-effector to a pose. Pre: hasProperty(Target, Posi-

[2]eBT ROS package: https://github.com/frovida/ebt.git

Algorithm 2: preProcessNode(s, node, wm, im)

```
   // Instanciate the procedure
1  while node.isAbstract() or not checkPreCond(node, wm) do
2      if not im.ground(node) then
3          return False
4      parametrize(node, wm)
   // Remove redundant procedure
5  if checkPostCond(node, wm) then
6      node.children().clear()
7      return True
   // The desired parent processor
8  processor = None
   // Move back the node front
9  while s.undo() do
       // Check lemma 1
10     if not checkRules(node, s.recall(), wm) then
11         s.redo()
12         processor = Serial
13         break
       // Check lemma 2
14     if s.recall().isParent(node) then
15         addParentCond(node, s.recall())
   // Heuristic: Swap with neighbour parent nodes
16 while s.recall() and (procedure.inSubtreeOf(s.recall()) or
     s.recall().isEnd()) do
17     s.redo()
18     processor=None
   // Insert in the sequence
19 s.addStart(node, processor)
20 if node.hasChildren() then
21     s.redoAll()
22 return True
```

Algorithm 3: postProcessNode(s, node, wm, im)

```
  // Remove redundant procedure
1 if checkPostCond(node, wm) then
2     return True
  // Move back the node end
3 while s.front()!=node and not s.front() in node.children() do
4     s.undo()
5 s.addEnd(node, processor=parallel)
6 s.redoAll()
7 return True
```

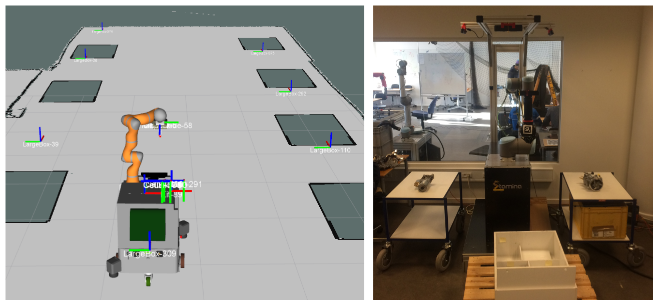

Fig. 3. The two robotic platforms used for the evaluation. The LH mobile robot used in simulation (left) and the static STAMINA robot for real pick-and-place (right).

tion). Post: (Gripper, at, Target)

- **plan_move(Initial, Target, Trajectory)** - calculates an arm trajectory between two poses. Post: isGrounded(Trajectory)
- **move_exe(Arm, Trajectory)** - execute an arm trajectory. Post: !isGrounded(Trajectory)
- **gripper_oc(Gripper)** - open and close the gripper
- **build_pose(Pose, Target)** - calculates and add an approach pose to the target. Post: isGrounded(Pose), (Target, hasA, Pose)
- **hold(Pose, Gripper, Object)** - Pre: (Gripper, at, Pose), (Object, hasA, Pose) Post: (Gripper, contain, Object)
- **release(Pose, Gripper, Object, Cell)** - Pre: (Gripper, at, Pose), (Cell, hasA, Pose) Post: (Cell, contain, Object)

The precondition *isGrounded(x)* holds by default for each param *x* of a skill, unless it is a postcondition of the skill itself. The pre and postcondition *property(= x[DeviceState] Idle)* holds for the hardware devices used by the skills. The domain has been slightly simplified here to respect space limits. The same skill set has been ported on two different robotics platforms (Fig. V) with a slightly different hardware configuration. Since the STAMINA robot is static, we implemented a fake *drive* skill - that takes no time to execute - to keep a comparable domain. For LH, we simulate the drivers of arm, gripper and mobile base, meanwhile we have a fake implementation of the *locate* and *registration* primitives, that add a new object in a fixed position in the container without localizing it with the camera. For LH navigation we used the standard ROS navigation stack [3], meanwhile for both robots we used moveIt [4] to plan and execute arm trajectories.

B. Results

The procedures are compared in Fig. 4, considering the case of a 1 part mission, with final return at the base station. The skill sequence (drive-pick-place-drive) is generated dynamically by a PDDL planner. On the upper part it is presented how the sequence would be expanded and executed normally. On the bottom, we have the equivalent optimized

[3]ROS navigation stack: http://wiki.ros.org/navigation

[4]moveIt: http://moveit.ros.org/

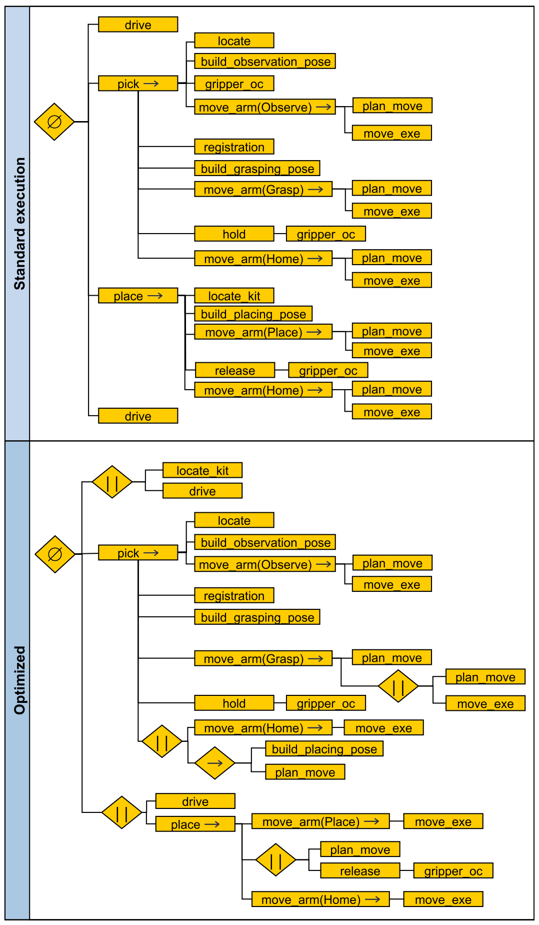

Fig. 4. Run-time optimization of an eBT. The initial eBT on the top, after processing with the optimization algorithm, is transformed into the new tree on the bottom, with several parallelized procedures.

eBT. We can see that many procedures are parallelized, in particular the locate_kit, the arm motion plan and the drive. Some redundant operations are filtered out (locate_kit, gripper_oc), when they have post-conditions already met. In some cases, this can be dangerous: for example, locating the kit multiple times can be advisable to refresh the information. In future work we plan to focus on a time aware optimization to balance better the sensing, based on the time of last update and the time available to the robot to sense. The results are presented in Table III. In the lab set-up, we could test maximum 2 parts missions, since the robot is static and has access to just two containers (right and left), meanwhile with the mobile robot we simulated up to a 6 part mission, a full kit. In general, considering the time spent on optimization, we can see that the time save varies between 11% and 20%. These timings are an average over 10 repetitions done for both the standard execution and the optimized execution.

VI. CONCLUSIONS AND FUTURE WORK

We presented the eBT model, a model to script robot skills that presents a good balance between intuitiveness and complexity necessary to describe, simultaneously, execution and effects into a planning domain. With this methodology

Mission/Robot	PDDL plan time (s)	Std. Execution time (s)	Optimization time (s)	Optimized execution time (s)	Total std. execution (s)	Total optimized execution (s)	**Time saved**
1 part/LH	3,8	52	4,1	40	56	48	15%
2 parts/LH	4,1	98	8	75	102	87	15%
3 parts/LH	5,2	160	12	116	166	164	19%
4 parts/LH	5,5	192	16	144	197	166	16%
5 parts/LH	10	231	20	168	241	199	18%
6 parts/LH	10	287	24	205	298	240	20%
1 part/STAMINA	2,4	48	5,1	35	51	42	16%
2 parts/STAMINA	3,1	99	12	75	102	91	11%

TABLE III

TIMINGS OF STANDARD EXECUTION COMPARED TO OPTIMIZED EXECUTION, AVERAGE OVER 10 TRIALS.

the designer gets a powerful, yet practical method to design autonomous robot skills with intrinsic planning capabilities. The hierarchical structure and the possibility of defining abstract skills and selecting automatically an implementation at runtime is one advantage for the system flexibility. Moreover, eBT maintains an explicit representation, that can be post processed in different ways. We proposed an algorithm able to optimize a skill sequence dynamically generated by a PDDL planner, and we presented on a real industrial use-case how this optimization can save up to 20% of the execution time w.r.t. a standard sequential execution. In general, the time saved can vary greatly depending on the skills and the specific implementation, nevertheless we showed that it can be substantial even with a limited skill set.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Commission's 7th Framework Programme under grant agreement no. 610917 (STAMINA) and Horizon 2020 Programme under grant agreement no. 723658 (SCALABLE).

REFERENCES

[1] O. Madsen, S. Bøgh, C. Schou, R. S. Andersen, J. S. Damgaard, M. R. Pedersen, and V. Krüger, "Integration of mobile manipulators in an industrial production," *Industrial Robot: An International Journal*, vol. 42, no. 1, pp. 11–18, 2015.

[2] F. Rovida, M. Crosby, A. S. Polydoros, B. Großmann, D. Holz, R. Patrick, and V. Krüger, *SkiROS - A skill-based robot control platform on top of ROS*, ser. 707. Springer International Publishing, 2017, vol. 2.

[3] R. Simmons and D. Apfelbaum, "A task description language for robot control," *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No.98CH36190)*, vol. 3, pp. 1931–1937, 2000.

[4] V. Verma, T. Estlin, A. Jonsson, C. Pasareanu, R. Simmons, and K. Tso., "Plan execution interchange language (plexil) for executable plans and command sequences." in *In Intl. Symp. on Articial Intelligence, Robotics and Automation in Space (ISAIRAS) Germany*, 2005, pp. 205–212.

[5] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty architecture for robotic autonomy," *Aerospace Conference, 2001, IEEE Proceedings.*, 2001.

[6] S. Bensalem and M. Gallien, "Toward a more dependable software architecture for autonomous robots," *IEEE Robotics and Automation Magazine*, pp. 1–11, 2009.

[7] R. Janssen, E. V. Meijl, D. D. Marco, R. van de Molengraft, and M. Steinbuch, "Integrating Planning And Execution For ROS Enabled Service Robots Using Hierarchical Action Representations," in *16th International Conference on Advanced Robotics (ICAR)*, 2013, pp. 1–7.

[8] F. Dvorák, R. Barták, A. Bit-Monnot, F. Ingrand, and M. Ghallab, "Planning and acting with temporal and hierarchical decomposition models," in *Proceedings of the 2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, ser. ICTAI '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 115–121.

[9] D. Kortenkamp and R. Simmons, "Robotic systems architectures and programming," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 187–206.

[10] E. Gat, "On three-layer architectures," in *Artificial Intelligence and Mobile Robots*. MIT Press, 1998.

[11] R. C. Arkin, *Behavior-based Robotics*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

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

[13] N. Nilsson, "Shakey the robot. technical report 323," *AI center, SRI international*, 1984.

[14] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM - A Cognitive Robot Abstract Machine for everyday manipulation in human environments," *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010*, pp. 1012–1017, 2010.

[15] C. Schlegel, A. Lotz, M. Lutz, D. Stampfer, and C. Vicente-chicote, "Model-Driven Software Systems Engineering in Robotics: Covering the Complete Life-Cycle of a Robot," *it - Information Technology*, vol. 57, no. 2, pp. 85–98, 2015.

[16] M. Klotzbucher and H. Bruyninckx, "Coordinating Robotic Tasks and Systems with rFSM Statecharts," *Journal of Software Engineering for Robotics*, vol. 1, no. January, pp. 28–56, 2012.

[17] P. Allgeuer and S. Behnke, "Hierarchical and State-based Architectures for Robot Behavior Planning and Control," in *Proceedings of Workshop on Humanoid Soccer Robots, IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, 2013.

[18] J. Bohren, "SMACH," 2014, http://wiki.ros.org/smach.

[19] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ogren, "Towards a Unified Behavior Trees Framework for Robot Control," *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[20] M. Colledanchise and P. Ogren, "How Behavior Trees Modularize Robustness and Safety in Hybrid Systems," *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, 2014.

[21] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[22] A. Ferrein and G. Lakemeyer, "Logic-based robot control in highly dynamic domains," *Robotics and Autonomous Systems*, no. July 2008, 2008.

[23] S. Magnenat, "Software integration in mobile robotics, a science to scale up machine intelligence," PhD thesis, École polytechnique fédérale de Lausanne, Switzerland, 2010.

[24] S. Balakirsky, Z. Kootbally, T. Kramer, A. Pietromartire, C. Schlenoff, and S. Gupta, "Knowledge driven robotics for kitting applications," in *Robotics and Autonomous Systems*, vol. 61, no. 11. Elsevier B.V., 2013, pp. 1205–1214.

[25] Z. Kootbally, C. Schlenoff, C. Lawler, T. Kramer, and S. K. Gupta, "Robotics and Computer-Integrated Manufacturing Towards robust assembly with knowledge representation for the planning domain de fi nition language (PDDL)," *Robotics and Computer Integrated Manufacturing*, vol. 33, pp. 42–55, 2015.

[26] F. Rovida and V. Krüger, "Design and development of a software architecture for autonomous mobile manipulators in industrial environments," in *2015 IEEE International Conference on Industrial Technology (ICIT)*, 2015.

[27] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002.