

Behavior Trees and State Machines in Robotics Applications

Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski and Swaib Dragule

Abstract—Autonomous robots combine skills to form increasingly complex behaviors, called missions. While skills are often programmed at a relatively low abstraction level, their coordination is architecturally separated and often expressed in higher-level languages or frameworks. State machines have been the go-to language to model behavior for decades, but recently, behavior trees have gained attention among roboticists. Originally designed to model autonomous actors in computer games, behavior trees offer an extensible tree-based representation of missions and are claimed to support modular design and code reuse. Although several implementations of behavior trees are in use, little is known about their usage and scope in the real world. How do concepts offered by behavior trees relate to traditional languages, such as state machines? How are concepts in behavior trees and state machines used in actual applications?

This paper is a study of the key language concepts in behavior trees as realized in domain-specific languages (DSLs), internal and external DSLs offered as libraries, and their use in open-source robotic applications supported by the Robot Operating System (ROS). We analyze behavior-tree DSLs and compare them to the standard language for behavior models in robotics: state machines. We identify DSLs for both behavior-modeling languages, and we analyze five in-depth. We mine open-source repositories for robotic applications that use the analyzed DSLs and analyze their usage. We identify similarities between behavior trees and state machines in terms of language design and the concepts offered to accommodate the needs of the robotics domain. We observed that the usage of behavior-tree DSLs in open-source projects is increasing rapidly. We observed similar usage patterns at model structure and at code reuse in the behavior-tree and state-machine models within the mined open-source projects. We contribute all extracted models as a dataset, hoping to inspire the community to use and further develop behavior trees, associated tools, and analysis techniques.

Index Terms—behavior trees, state machines, robotics applications, usage patterns, exploratory empirical study



1 INTRODUCTION

The robots are coming! They can perform tasks in environments that defy human presence, such as fire fighting in dangerous areas or disinfection in contaminated hospitals. Robots can handle increasingly difficult tasks, ranging from pick-and-place operations to complex services performed while navigating in dynamic environments. Robots combine skills to form complex behaviors, known as missions [1], [2], [3]. While skills are typically programmed at a relatively low level of abstraction (such as controllers for sensors and actuators), the coordination of skills to form missions is either programmed at a low level, intimately tied to the implementation of skills, or with higher-level representations using behavior models. With the increasing complexity of robot systems, higher-level representations of missions have become increasingly important to improve software quality and maintainability [4], [5], [6].

State machines are among the most common notations for describing behavior models in robotic missions [1], [7], [8]. Recently, behavior trees are attracting the attention of roboticists to express such high-level coordination. Both models describe pre-defined missions with limited decision making. Behavior trees were invented to model the behavior of autonomous non-player characters in computer games. Similar to autonomous robots, these are reactive and make decisions in complex environments [9], [10], [11].

Many researchers have observed that behavior trees can offer ease of reuse, modularity, and flexibility when modeling reactive behavior [7], [12], [13], [14], [15], [16], [17], [18], [19],

[20], [21], [22]. These works mainly focus on theoretical and proof-of-concept aspects of behavior-tree models. There is a trend in the robotics community to create domain-specific languages (DSLs) [23] and model-driven tools to engineer software for robotics systems [24], [25], [26], but we need to better understand and improve these solutions to make them usable in practice [25]. Multiple DSLs have been created to support the implementation of behavior-tree and state-machine models, but we lack studies to understand these implementations from a software-engineering perspective and their use in real-world projects.

Our study bridges this knowledge gap by studying the actual concepts offered in real implementations of behavior-model DSLs and their usage in practice. We compare DSLs for behavior trees, the emerging language in robotics, to state machines, the traditional choice of roboticists. In this paper, we explore the concepts offered by behavior trees and state machines as realized in five DSLs (internal and external DSLs offered as libraries) and compare them. The scope of our comparison is behavior-tree DSLs and their offered concepts. We further study how the language concepts are exploited by the users of the analyzed libraries, based on an analysis of their usage in open-source ROS projects. Our goal is to obtain empirical data on the use of these behavior-modeling languages in practice.

Goal and Research Questions

We present an exploratory study of behavior-tree and state-machine languages and their use in open-source robotic applications. While mostly qualitative, we provide quanti-

tative data about the models we mined, and indicate the frequency of different concepts and phenomena we observe in practice. We formulated the following research questions:

RQ1. *What modeling concepts from behavior trees and state machines are available in language implementations (libraries)?*

We identified five DSLs (libraries) for behavior trees and state machines that support ROS [27]—a middleware and framework for developing robotics applications, surrounded by the largest currently existing ecosystem of robotics libraries (ROS packages)—and are actively maintained and documented. Then, we identified and analyzed the concepts offered by these DSLs. Our goal was to understand the available behavior-tree and state-machine concepts in practice, and the similarities and differences between these.

RQ2. *How are these languages (libraries) engineered in practice?*

For the identified behavior-tree and state-machine DSLs, we studied the key design principles and language implementations. By studying the language implementations, we capture the techniques and practices that are used in the robotics community. By analyzing languages that originate from the practice of roboticists, we can understand the design concepts needed in robotics better and discuss potential improvements.

RQ3. *How are behavior-tree and state-machine models used in robotics projects?*

We mined GitHub for open-source repositories that use the identified behavior-tree and state-machine DSLs, and we analyzed their usage in robotics applications. We checked the usage trends of the languages, which can indicate their popularity. We extracted a sample of the mined projects and analyzed the usage of behavior-tree and state-machine concepts and the structure of the models in these projects. Finally, we investigated reuse mechanisms in the projects by a combination of visual and code-level inspections. In summary, we report on empirical results regarding the use of behavior trees and state machines in open-source ROS robotics projects in terms of the popularity of the DSLs, the structure of the models, the usage frequency of identified concepts, and the state-of-practice of reuse in behavior-tree and state-machine DSLs.

Journal Extension

This paper extends a study of behavior trees, presented at SLE 2020 [28]. Compared to that paper, we here broaden the scope of our study to include state machines. We extract state-machine modeling concepts and compare them to behavior-tree concepts extracted in the former work. We update the collection of open-source behavior-tree models and additionally mine open-source robotics projects that uses state-machine DSLs. We improve the selection process of included projects (filtration) and automate the process using scripts. The new filtration process is applied to all the mined projects. Finally, we qualitatively and quantitatively analyze a random sample of the mined state-machine models that matches the number of previously analyzed behavior-tree models, to understand the usage of the studied behavior-modeling languages in open-source projects. Thus, compared to the former paper, our focus is shifted from only behavior trees to also include a comparison with state machines in terms of concepts and robotics applications.

Results

Our analysis of the modeling concepts offered in behavior-tree and state-machine DSLs (**RQ1**) shows similarities in terms of language design and offered concepts. The languages are open and support domain-specific patterns, which reflect common needs in robotics. Openness is a common feature in the studied DSLs, which do not enforce a fixed model, but allow, even expect, concrete projects to extend the DSLs by-need. Another observation is that all studied DSLs offer constructs for frequent control-flow patterns. Although the range of support differs between behavior-tree and state-machine DSLs, accommodating the needs of the users of a specific domain is good language design practice.

We have observed in our analysis of the DSL design (**RQ2**) that having a visualization tool for model construction and monitoring could provide a better understanding of the models and improved code reuse. In the DSLs that feature a graphical notation with a GUI-based editor, projects were more often using built-in language constructs, such as Decorators in behavior trees and Concurrency containers in state machines (introduced in Section 4). Projects using DSLs without GUIs leaked these constructs to the code instead of the model. This is closely related to the fact that DSLs with no GUI are considered internal while DSLs with GUIs are external. External DSLs enforce the use of the DSL constructs (like decorators), while it is easy in internal DSLs to deviate and use ordinary programming-language constructs. Although it is a pragmatic practice, internal DSLs will hinder maintainability and analyzability of the behavior model in the long term. Another observation is that all the DSLs follow the *models-at-runtime* paradigm [29], [30]: models coordinate skills, actions, and tasks, which are implemented by lower-level means, such as ROS components.

Our analysis of the sampled models of robotics projects using behavior-tree and state-machine DSLs (**RQ3**) has provided insights into the usage of these languages in practice. First, the usage of behavior trees is rapidly increasing within open-source robotics projects. Second, the DSL usage among developers was similar from a structural perspective. Developers kept the structure of the models fairly simple; shallow and moderately sized models were observed in the majority of the sampled projects both for behavior trees and state machines. From our experience in analyzing the sampled models, keeping the behavior model simple, regardless of the type, helps in its understandability. Finally, we observed three code-reuse patterns in the sampled models, with similarities in how these code-reuse patterns were used. To reuse a skill (action), the main reuse pattern was inter-model referencing; to reuse a task (multiple actions combined in a sub-tree or state machine), the main reuse pattern was clone-and-own.

Perspectives

With this paper, we hope to raise the interest of two research communities—software language engineering and software modeling—in languages for robotics behavior. We hope that observing the current state-of-practice can help to improve on it. We also hope that this analysis can inspire designers of behavior-tree and state-machine languages to revisit, or at least justify, some design choices. In addition, it seems

beneficial to take improvement points from each other, since some of these tools are built with model-based design concepts, and other good language design principles in mind.

Finally, we contribute a dataset of open-source behavior models, to inspire the community to use and further develop these languages, associated tools, and analysis techniques. An accompanying online appendix [31] contains these models, our mining and analysis scripts, and further details.

2 BACKGROUND

In the robotics community, different control structures are used to coordinate agent behavior, including behavior trees, state machines, teleo-reactive architecture, subsumption architecture, sequential behavior composition, flowcharts, and decision trees [7], [8], [32]. Each of these structures has its own advantages and disadvantages [7], [32].

Many of these control structures are offered as DSLs (domain-specific languages) [23] to developers. For mobile robots, many of these DSLs are even end-user-oriented, offering a visual syntax, see our previous study [33]. There, we observed that most of these DSLs are still imperative at a relatively low level of abstraction, similar to programming languages. Interestingly, we observed that many of these DSLs are realized by cutting down a real programming language and implementing a visual syntax for the remaining language concepts: expressions, declarations, and statements (including robot-specific library method calls). The visual syntax is often realized using Scratch or Blockly, offering a simple, block-based programming interface. These end-user-oriented DSLs target technically skilled users, while the other DSLs, especially the behavior-tree and state-machine DSLs we study in the sequel, clearly target developers.

2.1 Behavior Trees

In recent years, behavior trees have become a popular behavior-modeling language to specify missions and coordinate their control flow. Behavior trees have been shown to generalize multiple control architectures [16], [34]. State machines have been the traditional go-to model in robotics for their simplicity and their easy-to-understand control-switching mechanism, but they become hard to maintain when missions grow in complexity [8], [15]. Behavior trees are claimed to overcome this obstacle, due to their modularity and flexibility [7], [15], [16]. Missions specified using behavior trees can be mapped into state machines and vice versa [16], [18]. In contrast, in this paper we study their use in real robotics projects, which has not been done before.

Applications of Behavior Trees

Behavior trees are well-suited to express the runtime behavior of agents, which has fueled applications in computer games and robotics. High-profile games, such as Halo [10], use behavior trees. However, the implementations of behavior trees in gaming differ from their implementations in robotics. For example, the *Unreal Engine 4 (UE4) Behavior Tree*, probably the world’s most used behavior-tree DSL, emphasizes event-driven programming rather than time-triggered control, which is a major concern in robotics.

In the robotics community, the interest in behavior trees is currently growing. Hierarchical state machines

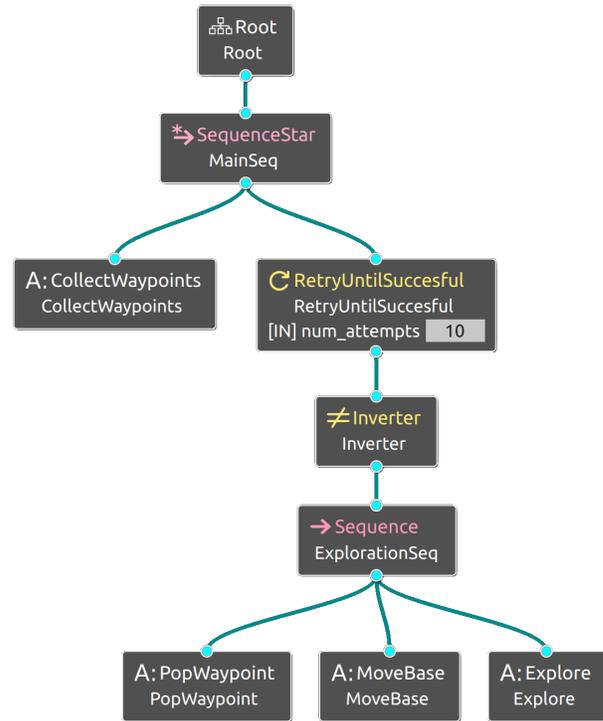


Fig. 1: A behavior tree of a health and safety robot inspector from a GitHub project `kmi-robots/hans-ros-supervisor`, shown in the Groot editing and animation tool from `BehaviorTree.CPP`.

were the main task-orchestrating mechanism in the ROS navigation system supported by different languages [35]. With the upgrade to the newer version of ROS, ROS 2, the main customization mechanism navigation was changed to behavior trees [36]. ROS 2 still supports hierarchical state-machine languages [37], but they do not use them as the main customization mechanism. Further witnessing the increasing popularity of behavior trees, IROS’19, one of the key research conferences in robotics (behavior-trees-iros-workshop.github.io). In addition, multiple projects in RobMoSys (robmosys.eu), one of the leading model-driven communities in robotics, have been launched to create a set of best practices and tools for behavior trees, such as CARVE (carve-robmosys.github.io) and MOOD2Be (robmosys.eu/mood2be). The EU project Co4Robots (co4robots.eu) developed a mission-specification DSL for multiple robots based on behavior-tree concepts [21], [38]. Finally, behavior trees are used for autonomous-driving systems. Autoware [39], a leading open-source platform for self-driving vehicles, adopted behavior trees for the coordination between the supported pre-defined driving scenarios (autowarefoundation.github.io/autoware.universe/main/planning/behavior_path_planner/#behavior-tree). CARLA [40], a high profile open-source simulator for autonomous driving research, uses a behavior-tree DSL, `PyTrees`, which is among the DSLs we study in the paper, to define non-ego vehicle behavior [41].

Behavior-Tree Example

Figure 1 presents an example of a behavior-tree model of a health and safety inspector robot from the Knowledge Media

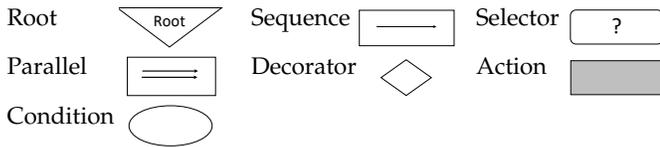


Fig. 2: Node types in behavior trees (visual syntax)

Institute (kmi.open.ac.uk). The robot performs an exploration sequence for an area. The main operation is placed at the bottom, in the sub-tree under `ExplorationSeq`: it consists of obtaining the next waypoint, moving the mobile base to the waypoint, and exploring the area. If obtaining a new waypoint fails (empty stack), the first task fails, which is converted into a success by an `Inverter` such that the sequence of motions has been completed. Otherwise, the robot keeps repeating the same operation (next waypoint, move, explore) up to ten times, as long as the stack is not empty. The entire computation is placed in a loop that alternates between obtaining new waypoints and performing the exploration sequence (`MainSeq`) until the success of all children.

Behavior-Tree Concepts

In general, behavior trees can be seen as graphical models that are shaped as directed trees, with a dedicated root node, non-leaf nodes called *control-flow nodes*, and leaf nodes called *execution nodes*. A behavior tree is executed by sending signals called *ticks* from the root node, traversing the tree according to the semantics of the control-flow nodes. Ticks are issued with a specific frequency [7], [42]. Upon receiving a tick, a node executes a task, which can be a control-flow task or, if a leaf node is ticked, some specific robotic task (a.k.a. skills). The latter classifies into actions (e.g., `MoveBase` in Fig. 1) and conditions, which can test propositions (e.g., whether the robot is at its base) used to control task execution. A ticked node returns its status to its parent: (1) *success* when a task is completed successfully, (2) *failure* when a task execution failed, and (3) *running* when a task is still under execution.

This execution semantics is interesting and can be seen as the main difference to other behavior-modeling languages. As we will discuss for our concrete behavior-tree DSLs in the sequel, the current state of execution is not explicitly represented in behavior trees. In most other languages, such as state machines or flowcharts, the state advances (as prescribed by the control-flow elements, such as transitions) by moving the current point of execution (the state) forward. Consider, for instance, flowcharts, where one action is executed after another, controlled by control-flow elements between the actions. Notably, behavior trees are also actions controlled by control-flow elements. In other words, behavior trees are like many other imperative programs—recall also the many end-user-oriented robot-control DSLs studied in our previous work [33], most of which are subsets of imperative programming languages with a visual syntax represented in Scratch or Blockly. Now, in behavior trees, the tree is re-executed regularly in short time intervals, which allows to re-execute actions. Intuitively, one can use that to put reactive control more to the upper-left of the tree, where it gets regularly re-executed by every tick (determined by a control loop), while more deliberative skills (actions) are more to the bottom and right of the tree. Adding such reactive control to other languages, including

state machines and flowcharts, is not easily possible, not even with hierarchical state machines. In other words, this special execution semantics fosters modularity and allows to unify other control architectures, such as the subsumption architecture [16].

The benefit of using behavior trees lies in their ability to express task coordination using a small, but extensible set of control-flow nodes. Most behavior-tree languages offer the following control-flow nodes: sequence, selector, decorator, and parallel. The example in Fig. 1 illustrates two sequence nodes (`MainSeq` and `ExplorationSeq`) and two decorator nodes (`Inverter` and `RetryUntilSuccessful`). Intuitively, sequence nodes tick all children and require that they all succeed for the sequence node to succeed, while selector nodes only require one child’s success to succeed. Decorator nodes allow more complex control flow, including for- and while-loops. They are also extensible; developers can implement custom decorator nodes. Parallel nodes are a misnomer. They are generalizations of sequence and selector nodes, allowing custom policies, such as cardinalities specifying the minimum or maximum number of nodes that need to succeed. In the literature, there is no clear agreement if parallel nodes execute child nodes simultaneously, as the name suggests, or concurrently [7], [11], [32], [42], [43], [44].

The visual presentation of the main node types, as illustrated in studies of behavior trees in robotics and games [7], [42], [44], is summarized in Fig. 2. Our example in Fig. 1 shows the visual syntax used in a popular behavior-tree DSL called `BehaviorTree.CPP`, which is among the DSLs we study in the remainder of the paper. Although the shapes of the nodes might differ between publications and actual language implementations, the inner symbol of each node-type is usually the same; e.g., the selector node always uses the symbol “?”.

2.2 State Machines

State machines are probably the most popular and well-researched language for behavior specification of systems. They provide the basis not only for modeling, but also for analysis (e.g., for model checking) or synthesis of software controlling real systems. There is a wealth of literature on state machines in non-robotics contexts. As such, we do not discuss the applications in detail here, as opposed to behavior trees above, but to further illustrate behavior trees in the well-known state-machine syntax, we provide an example and then recap the main state-machine concepts again.

State-Machine Example

Figure 3 shows the same mission as in Fig. 1 for a health and safety inspector robot, represented as a state machine, using the notation of the Unified Modeling Language (UML) (see Fig. 4 for the UML syntax). Since state machine DSLs offer a rich notation for modeling behavior, there is no single mapping from behavior trees to state machines. We modeled the actions and the control flow in Fig. 1 as follows.

States represent some situation that the modeled system is in for some period of time. From the actions in our behavior tree, we model the collection of way points and the exploration sequence as states, where the latter consists of the states `MoveBase` and `Explore`. These states have the

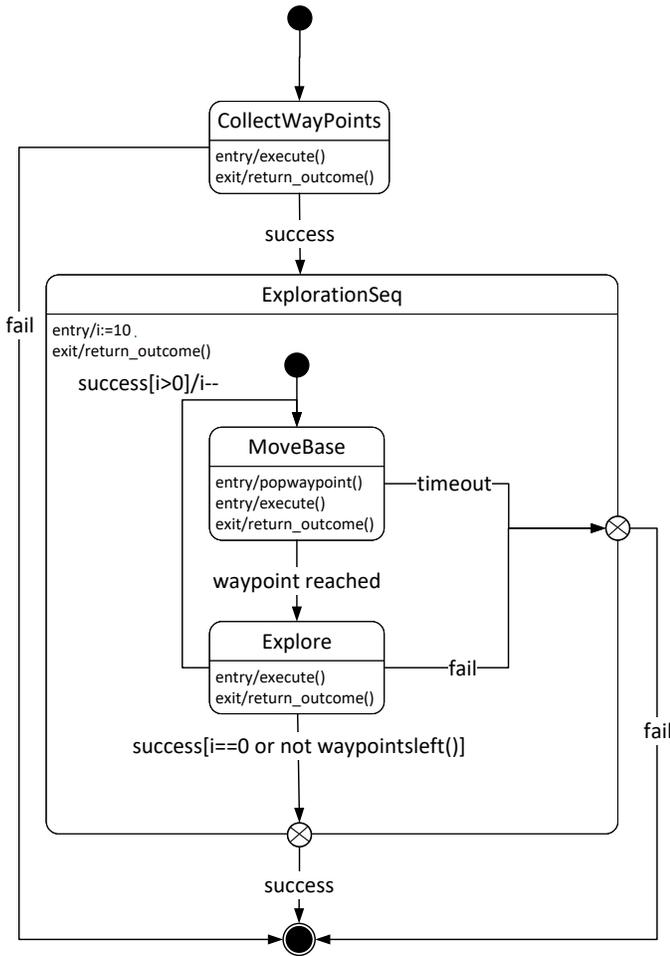


Fig. 3: A state machine representing the same mission as in Fig. 1, using UML syntax.

respective actions from the behavior tree, executed on state entry or state exit. In contrast, the action `PopWaypoint` is rather atomic, so we do not represent it as a state, but execute it when the state `MoveBase` is entered.

The `ExplorationSeq` of the behavior tree model is placed in a nested, also called composite, state `Exploration_seq`. At the entry of `Exploration_seq`, a variable `i` is set to the desired number of attempts (here, ten). For each waypoint, the `MoveBase` component is activated and the model waits in the state until the sensors and the localization component detect that a waypoint has been reached (or a timeout occurs). The model exits from `Exploration_seq` if exploration has achieved its goals (the ten attempts are finished; the triggering event is success and the guard's condition is `i==0`). If the event success is triggered from `Explore` and `i>0`, the counter `i` is decreased at the transition from `Explore` back to `MoveBase` (which is an alternative to having actions on state entry or exit in UML).

Finally, transitions are triggered by events, which are signals coming from the actions or from the environment. In behavior trees, the control-flow is determined by the actions' outcome, which is either success or fail. In our state machine, we also used success or fail as the events originating from action results for some transitions. For others, we used more domain-specific events. For instance, `timeout`

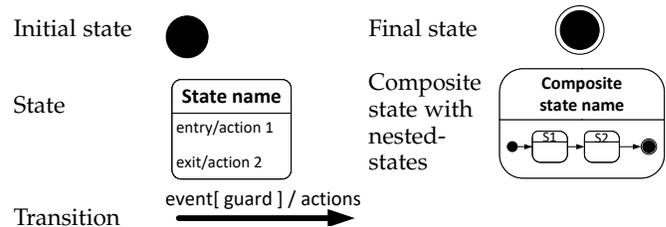


Fig. 4: Node (state) types and transitions (visual syntax) for state machines, according to UML.

is an external event from the system, which ends the state `MoveBase` and then the whole system; alternatively, the event `waypoint reached` moves the system to the state `Explore`, where the exploration either continues with the next waypoint or also ends the whole system. If no waypoint is left (indicated in the guard on one of the state's leaving transitions), then `popwaypoint()` would fail. Since in the behavior tree, this would be handled by the dedicated action, and we have no specialized state for it, we utilize a method `waypointsleft()` to avoid going into `MoveBase` again.

State-Machine Concepts

State machines and hierarchical state machines have been used for decades in different domains with slight variations in syntax and semantics [45], [46]. While there exist multiple syntaxes with minor variations, the UML's state diagram, a graphical representation of state machines in UML, might be the most common visual syntax, especially in the software modeling community (see Fig. 4).

A state-machine model is a directed graph of three primary elements: states, transitions and actions. In the UML semantics [45], [47], [48], states represent patterns of behavior. The behaviors executed by the robot are represented as actions. From the perspective of computational execution semantics, UML's state machines support both the execution semantics of Mealy and Moore machines [49], [50]. When an action is associated to a transition (action-on-transition), similar to Mealy machines, the action is executed at the transition. When the action is associated to a state (action-on-state), similar to Moore machines [51], the action is executed at the state entry or exit. Most state-machine implementations are event-driven.

Hierarchical state machines, also known as statecharts [52], are an extension of state machines that allow nesting (hierarchy) and concurrency [52], [53]. Thus, hierarchical state machines are similar to state machines but can additionally separate behaviors into sub-states, or composite states, thereby increasing the modularity of the model and decreasing the number of transitions needed.

Just like behavior trees in robotics, state machines provide graphical models to represent the behavior of a robotic agent. Colledanchise and Ögren [16] have shown how behavior trees generalize state machines in robotics. In the rest of the paper, we focus on implementations in robotics; the robotic implementations of state machines will differ slightly from the UML notation.

3 METHODOLOGY

We now describe our methodology to (RQ1) identify relevant DSLs and the concepts offered by behavior trees and state machines, to (RQ2) analyze their implementations, and to (RQ3) identify and analyze open-source robotics projects using these DSLs.

3.1 Identifying DSLs and Language Concepts (RQ1)

To identify behavior-tree DSLs, we searched on GitHub for projects with behavior trees in robotics using different search terms, including “behavior tree robotics” and “behavior tree robot.” The search returned projects using different behavior-tree DSLs. We identified the imported DSLs in the projects. We focused on DSLs in Python and C++, the most used programming languages in robotics. We also looked at the behavior-tree literature in robotics to identify DSLs.

A similar search for state-machine DSLs yielded hundreds of results on GitHub. Therefore, we decided to use the ROS wiki (wiki.ros.org) to find state-machine DSLs. The wiki is commonly used by developers to publish their open-source languages and tools for developing robotics applications. By searching it instead of GitHub or Google, we ensured that the identified languages support ROS, so we excluded non-robotics projects in later steps.

To ensure the relevance of the identified DSLs (offered as libraries) for real-world robotics applications for both behavior trees and state machines, we focused on maintained libraries and for that we applied the following *exclusion criteria*: (1) lack of documentation, (2) out-dated libraries not maintained anymore (last commit older than 2019), (3) no ROS support (this was checked specifically for the identified behavior tree DSLs since we collected them through GitHub) and (4) no mined projects.

Thereafter, to understand the key characteristics and modeling concepts offered by the included DSLs, we identified their main language constructs and studied the relation between them. For our comparison, we collected behavior-tree concepts by an exploratory literature search [7], [32], [44] and snowballing. For the identified libraries, we inspected their documentation [54], [55] and wrote scripts that execute small tests to understand better their semantics. A similar approach was followed to collect state-machine concepts from the literature [56], [57], [58], [59] and the documentation of the identified libraries [60], [61]. In this process, we focused on behavior-tree concepts and whether state machines offer direct support for similar concepts or they need to be expressed indirectly. Our analysis was iterative, to ensure a proper reflection of the concepts in the different DSLs. We scoped the comparison to concepts offered by the included DSLs that met our criteria.

3.2 Analyzing DSL Implementations (RQ2)

After identifying relevant libraries that support state-machine and behavior-tree modeling languages and analyzing their syntax and semantics, we wanted to understand the implementation design of the libraries. We built our findings from inspecting different sources. Specifically, we inspected the implementations of the libraries on GitHub, documentations [54], [55], [60], [61], [62], tutorials [63], [64], [65], [66], and related publications [56], [57], [59], [67], [68], [69], [70].

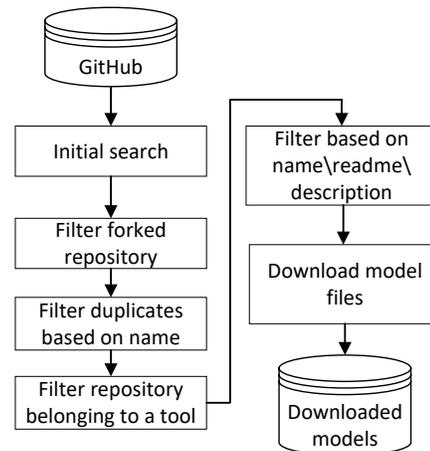


Fig. 5: Filtering steps to identify relevant repositories.

We focused on the language design of the libraries and the concrete syntax offered. We went through the tutorials of the libraries and related publications to reflect on the dynamicity of the libraries. By dynamicity, we refer to the ability of runtime modification of models. Finally, we examined the models of concurrency used in the libraries by examining their implementations on GitHub and the documentation.

3.3 Identifying and Analyzing Robotics Projects (RQ3)

To understand the usage of the identified DSLs in robotics projects: (1) we mined GitHub for open-source repositories using these languages in their implementation of robotics missions, then (2) we compared the popularity of the DSLs in open-source projects over time and analyzed a sample of these projects in terms of the usage of the languages concepts and the structure of the models, and finally (3) we observed how the DSLs users have carried out code-reuse in practice. In the sequel, we describe these steps in detail.

Mining GitHub

Open-source projects have been used in software-engineering and robotics research communities to understand real-world applications. Different open-source platforms, such as GitHub, provide great opportunities for researchers to identify the state-of-practice in robotics and software engineering [71], [72], [73]. Motivated by our previous work [28] and other researchers who mined GitHub, we used GitHub as a source for mining open-source robotics projects.

For the identified behavior-tree and state-machine DSLs, we investigated how they are used in the source code of robotics projects. In `BehaviorTree.CPP`, the term `main_tree_to_execute` refers to the entry point tree in the XML source code, while the term `py_trees_ros`, `smach_ros`, and `flexbe_core` are used to import the languages `PyTrees_ros`, `SMACH` and `FlexBe`, respectively. These terms must be used in the source code of the targeted languages. We created a Python script to mine GitHub repositories for those terms with a simple text-match in source code, using GitHub’s code search API (github.com/PyGithub/PyGithub).

Next, we wanted to filter-out projects belonging to a course or tutorial, to better reflect on actual usage of the libraries in robotics. Inspired by the work of Malavolta

et al. [71] in providing mining guidelines for robotics software, we used a similar filtering mechanism with minor changes to adopt it to our goal. Figure 5 shows our filtering mechanism. After mining GitHub for open-source projects (step 1), we excluded forked repositories using a Python script (step 2). By filtering forked repositories, we excluded duplicate models. After a quick inspection, we noticed that there are duplicate repositories that were not forked, instead they might have been cloned and re-uploaded to the user’s GitHub account. To filter them out, we extracted the repositories with the same project name belonging to different GitHub users using a Python script, inspected them manually to ensure they are duplicates, and deleted them from our lists (step 3). In our previous inspection, we noticed that repositories with the keyword (tool) in their name often belonged to software tools where the targeted libraries are requirements for their functionalities. Since our goal is collecting the implementations of robotics projects rather than supportive tools, we excluded repositories with the keyword (tool) in their name using a Python script (step 4). We also excluded the following organizations as users, since they either correspond to tutorials repositories of the library creators, or they are known tools using the identified languages as requirements (BehaviorTree, splintered-reality, team-vigir, ros-planning, ros-infrastructure, carla-simulator). Finally, we used different Python scripts to inspect the name, readme, and description of the repositories, and exclude those with the following keywords: (assignment, course, tutorial, introduction) (step 5). This ensured the exclusion of projects belonging to (1) an assignment or (2) a course. By the end of this step, we had a list of relevant repositories that match our criteria.

Our next step was to download the files that contain the models from these repositories (step 6). Through our analysis while extracting concepts in Section 4, we identified specific terms used in each DSL when constructing the behavior model. In `BehaviorTree.CPP`, the term `main_tree_to_execute` refers to the entry point when constructing the tree. In `PyTrees_ros`, `add_child` is usually used when adding nodes to the tree. Finally, `StateMachine.add` and `OperatableStateMachine.add` are used in `SMACH` and `FlexBe`, respectively, to add states. Using these terms, we matched the files in the GitHub repositories that contain them and downloaded them. For all the previous steps, we used Python regular expressions, Python’s Requests API, and the code-search API from GitHub. All code can be found in our online appendix [31].

Analyzing Models

To understand the usage of behavior trees and state machines in open-source robotics applications (RQ3), we analyzed the mined projects from two different perspectives: the popularity of the DSLs over time and the structure of models. As a start, to understand the popularity of the DSLs, we extracted the creation and last commit dates for the repositories to plot the number of active projects per year. We assume a project is active in the duration between its creation date and last commit. We used all mined projects before filtering and only excluded those belonging to the DSLs’ organizations/creators, which are the following: BehaviorTree, team-vigir, pschillinger, FlexBE, splintered-reality,

ros, ros-visualization. The collected projects were mined until 31/12/2021 to capture activity until the end of 2021.

Moving to the analysis of the models, we were interested in understanding the usage of behavior trees and state machines in real robotics projects. Consequently, we used the list of relevant repositories from the filtering step and their downloaded models. At the time of the analysis of the behavior-tree models, we got 75 models. Although an updated mining at the beginning of 2022 to cover the projects until the end of 2021 yielded new models and projects, we decided to keep the same sample, since it covers different sizes of models and domains. For state-machine models, we randomly sampled 75 models to match the number of behavior-tree models to have comparable results. The state-machine mining yielded thousands of models; consequently, we randomly sampled projects. We defined two project pools according to the model size (defined as number of nodes, see below for metric details), one for normal size [2–6] and another for large size [7–66], then we randomly sampled from each pool a number of projects using the sampling API (pandas.pydata.org/docs/reference/api/pandas.DataFrame.sample.html) from the Python DataFrame library. The model size range for each pool was decided based on the data distribution. The total sample size corresponded to the number of models belonging to behavior-tree projects meeting our repository inclusion criteria (75 models).

We calculated metrics that capture the core structural aspects of the models and report on the usage of behavior-tree and state-machine concepts. Since we are analyzing two different types of structures, directed cyclic graphs in state machines and directed acyclic trees in behavior trees, it is challenging to use similar metrics. Thus, we calculated some common metrics between the two architectures and other metrics only for one architecture. To distinguish between them, we are using the abbreviations SM and BT to indicate which architecture metric belongs to. The metrics are:

- Model size (BT.size, SM.size): the total number of nodes excluding the root node in behavior trees and the total number of states in state machines.
- The tree depth (BT.depth): the number of edges from the root node to the deepest node of the tree [74]. Considering the example in Fig. 1, the behavior tree model has a BT.depth of five.
- Average branching factor (BT.ABF): the average number of children of each node. Considering the example in Fig. 1, the behavior-tree model has a BT.ABF of 1.6.
- Nesting level (SM.nesting): the composition or nested hierarchy level induced by counting the number of levels within the state-machine compositions [75], [76], [77]. We consider the main state machine as level one. In the example in Fig. 7, this state-machine model has a nesting level of three; where the main container `Inspect_SM` is level 1, the nested container `Iterator_10_attempts` is level 2 and the `Exploration_seq` is level 3.
- Node type percentage (N.pct): the frequency of a node type with respect to the total number of nodes. This metric captures the usage of the different composite node types in behavior trees and the different container constructs in state machines.

The extraction of these metrics relies heavily on the

implementations of the DSLs, so we used different extraction methods for each language. For behavior-tree DSLs, we inspected the code of both the tutorials of the libraries and a randomly sampled subset of mined models to understand how users implement them. We noticed that users follow different implementation styles, and that the models tend to be deeply intertwined with the rest of the code. While some metrics could be calculated automatically, for others, the models needed to be manually extracted. To calculate BT.size and N.pct, we extracted a function name for each node type based on the documentation of the libraries, then used a Python script to count the number of text matches. For leaf nodes, no automatic counting was possible since the libraries do not impose a specific implementation structure. We counted and calculated the percentage of leaf nodes, BT.depth and BT.ABF manually from the manually extracted models. To extract behavior-tree models, we were able to use a visual editor shipped with one of the identified libraries (Groot for BehaviorTree.CPP, explained shortly), where the behavior-tree language is realized as an external DSL. The other identified library (PyTrees_ros, explained shortly) constituted an internal DSL, where we needed to manually extract the model from the source code by identifying the respective library API calls used to construct the model. We considered every tree with a root node as a behavior-tree model.

Moving to state-machine DSLs, we also checked the code of tutorials of the libraries and a random sample of mined models. An implementation pattern was clear for each library, hence an automatic extraction of the metrics was possible. Using Comby (github.com/comby-tools), a parser that detects syntax in code based on user-supplied patterns [78], we defined a syntax pattern to match the start and end of a state-machine container to isolate it from the rest of the code. Then, we wrote a Python script to extract the state-machine model from the rest of the code. After extracting the model, another script was used to count the number of states and composite states to calculate SM.size and N.pct. Similar to the composite nodes count in behavior trees, a text match for the function name was used to count the different container types. Finally, we wrote a script for visualizing the extracted model to facilitate model inspection using SMCat (github.com/sverweij/state-machine-cat). We could not use the viewers provided by FlexBe and SMACH, because the models are intertwined with the code, and each project has different dependency requirements to run it. It was easier to isolate the model from the projects and visualize it using an external viewer. We wrote a Python script to transfer the extracted model code into the SMCat syntax and generate SMCat files. SMCat was executed on those files to generate SVG format files that visualize a corresponding state-machine model. Extracting the SM.nesting required manual work. Using the extracted visualized models, we went through all models and counted the nesting levels. All scripts and SVG files can be found in our online appendix [31].

Analyzing Reuse

The final aspect we analyzed is reuse, because it is one of the major issues in robotics software engineering [1], [6], [79], [80], [81], [82]. Reuse mechanisms are important to scale and sustain the use of behavior trees and state machines in practice. We analyzed the current state-of-practice for reuse

TABLE 1: Behavior-tree and state-machine DSLs identified. We analyzed those in bold

name	target language	ROS	doc.	last update
BehaviorTree.CPP github.com/BehaviorTree/BehaviorTree.CPP	C++	yes	[55]	2021/05
PyTrees github.com/splintered-reality/py_trees	Python	no	[54]	2021/05
PyTrees_ros github.com/splintered-reality/py_trees_ros	Python	yes	[62]	2021/05
BT++ github.com/miccol/ROS-Behavior-Tree	C++	yes	[83]	2018/10
SkiROS2 github.com/RVMI/skiros2	Python	yes	[84]	2020/11
pi_trees github.com/pirobot/pi_trees	Python	yes	n/a	2017/10
Beetree github.com/futureneer/beetree	Python	yes	n/a	2016/03
SMACH github.com/ros/executive_smach	Python	yes	[60]	2020/05
FlexBe github.com/team-vigir/flexbe_behavior_engine	Python	yes	[61]	2020/12
SMACC github.com/reelrbtx/SMACC	C++	yes	[85]	2021/05
RSM github.com/MarcoStb1993/robot_statemachine	C++	yes	[86]	2020/03
Decision making github.com/cognitsteam/decision_making	C++	yes	n/a	2016/07

in the studied models. By reuse we refer to reusing the code of a robotic skill (also known as action) instead of writing a new skill from scratch, or reusing the code of a repeated task (composed of different skills) in the same model or in a different one.

We observed reuse in the sampled projects using a mixture of visual and code-level inspection to detect any reused skill or task. We inspected the reuse of a skill or task in the model and across the different models of a project. A task is usually expressed as a sub-tree in behavior trees and as a composite state in state machines. A skill is expressed as a leaf node in behavior trees and as an action associated to a state in state machines. By first checking the model visualization, we could detect task-level and/or skill-level reuse in a model. We also relied in the task-level reuse on similar model structure in case of a slightly different combination of skills with a resembling structure. A deeper inspection of the code-level implementation of models and skills for the reused tasks and skills follows to understand what type of reuse mechanism is used.

4 LANGUAGE CONCEPTS (RQ1)

The search we conducted yielded 12 DSLs for implementing behavior-tree and state-machine models in robotics applications using ROS. The upper part of Table 1 lists the identified behavior-tree DSLs. We focus on analyzing the DSLs in the first three rows, set in bold font, for the following reasons. Among the DSLs relevant for robotics, these three

TABLE 2: Key concepts in our behavior-tree languages compared to our state-machine languages

behavior-tree concept/aspect	behavior-tree languages PyTrees, PyTrees_ros, BehaviorTree.CPP	state-machine languages SMACH, FlexBe
programming model	Synchronized and asynchronous are supported, time-triggered, activity-based. Reactive programming can be implemented to an extent using the tick concept and re-ordering of sub-trees.	Asynchronous, event-triggered, reactive
simple nodes	Execute actions (arbitrary commands, both instantaneous and long-lasting) or evaluate conditions (value translated to Success/Failure).	Basic state with associated action/conditions
exit status	Each node reports success, failure, or an in-operation state (“running”) each time it is triggered. Status report causes the computation (the traversal) to advance to the next node.	Each state reports its status (also known as outcome). Outcomes are user defined. Execute function checks periodically for outcome
composite nodes	Define hierarchical traversal, the control-flow for each epoch (tick). Sequentially composed. Nodes may start concurrent code though.	Similar to container concept. Allow state nesting and control-flow constructs.
root	Serves as entry point for every traversal. Has exactly one child node. Root node is re-entered at every epoch.	Initial state (first state added to the container).
sequence	Trigger children in a sequence until the first failure. If no failure return success, otherwise fail.	Supported using a predefined container.
selector	Trigger children in a sequence until the first success. If no success return failure, otherwise succeed.	No direct support, could use container to extend.
parallel	Generalize sequence/selector with a policy parameter. Several policies available, e.g., meeting a minimum number succeeding children.	No direct support, could use guards to extend.
goto (jumps)	No general jump construct, the computation always traverses the tree.	Supported.
decorators		
inverter	Invert the Success/Failure status of the child	No direct support, could use container to extend.
succeed	Return success ignores the status returned by the child	No direct support, could use container to extend.
repeat	Trigger the child node a set number of times, then succeed. Fail if the child fails.	Could use the <i>Iterator container</i> in SMACH as repeat-until loop where the desired outcome for breaking the loop is user-defined.
retry	Run the child node and retry it immediately if it fails for a maximum number of times, otherwise succeed.	Similar to repeat concept, <i>Iterator container</i> can be modified and used.
dynamicity	Runtime modifications of model (node re-ordering) possible due to the dynamic nature of the implementation.	Runtime modifications of model is possible (adding and removing states, changing state instantiation, Re-ordering states).
openness	New nodes and operators implemented by users as needed.	States and new control-flow is possible using the container concept.
concurrency	interleaving and co-routines; declared via sequence, selector, and parallel nodes.	interleaving; declared via concurrency container; states are executed sequentially but not parallel due to using Python threading.

were actively developed when we checked (2021/06/09). Together, they support ROS systems implemented in Python and C++, the two most popular programming languages in the robotics community. The latest version of ROS, ROS 2 [87], is supported by the included DSLs. PyTrees, the main behavior-tree DSL in the Python community, does not directly target ROS, but robotics in general. A popular extension, PyTrees_ros, provides bindings for ROS. Since PyTrees and PyTrees_ros are similar, with the only difference of ROS packaging, we decided to include PyTrees in the language analysis even though it does not support ROS directly. Although we analyzed both PyTrees and PyTrees_ros, in the remainder of this section we only refer to PyTrees_ros in the analysis for brevity, since our findings apply to both of them.

We decided to discard the remaining DSLs from our analysis. BT++ is now obsolete, superseded by BehaviorTree.CPP after the developer of BT++ joined the

latter as a contributor. SkiROS2 is a software platform for robotic task-level programming that uses design concepts from MDE and behavior trees as an execution engine. SkiROS2 is a new version of SkiROS [88], [89], which is now obsolete. SkiROS2 does not have open-source projects, so we needed to discard it from further analysis. Beetree and pi_trees are inactive experiments, now abandoned.

The lower part of Table 1 lists the state-machine DSLs we identified. We analyzed the first two (in bold font) and discarded the other three, since they match our exclusion criteria (cf. Sect. 3.1). SMACC is an actively maintained DSL that supports ROS systems implemented in C++. Robot Statemachine (RSM) is another C++ DSL with ROS support that also offers a GUI. Due to a lack of open-source projects on GitHub using SMACC and RSM, we needed to discard them. Decision making supports ROS systems in C++, but is no longer maintained. The two included DSLs are actively maintained when we checked (2021/06/09), have good

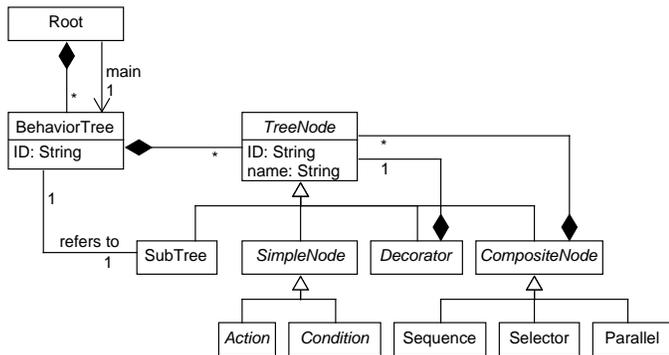


Fig. 6: A meta-model for BehaviorTree.CPP (reverse-engineered from its XML format)

documentation, support ROS systems in Python, and have multiple available projects on GitHub for mining. FlexBe supports ROS2, while SMACH does not [37].

We now reflect on the language concepts offered by the included behavior-tree and state-machine DSLs, both at the syntactic and semantic levels. The language concepts offered by our behavior-tree and state-machine DSLs are summarized and compared in Table 2. The left-most column names concepts that are pertinent to the behavior-tree DSLs, either due to inclusion or a striking exclusion from the behavior-tree DSLs. The last column comments briefly on how the respective concepts are handled in the state-machine DSLs. The remainder of this section discusses the details, first for behavior trees and then for state machines.

4.1 Concepts and Semantics in Behavior-Tree DSLs

The following discussion is based on a broad description of behavior-tree languages extracted from the available literature and documentation of PyTrees_ros and BehaviorTree.CPP [54], [55]. Table 3 presents the classes of the basic behavior-tree concepts in the analyzed DSLs.

The variant of behavior trees used in the analyzed DSLs is predominantly a *time-triggered activity-based* behavioral modeling language, unlike the implementation of behavior trees in gaming (see Sect. 2). The computation consists of activities that have duration and the main control loop triggers the entire model at (typically) fixed intervals of time like a circuit. Every tick (or epoch) triggers a traversal of the entire tree, with diversions introduced by the various types of nodes. The traversal can start new activities, evaluate conditions, access state, and execute basic actions for side effects. Reactive programming seems not to be supported first-class, despite reappearing statements to the contrary,¹ but can be simulated by sufficiently high-frequency model execution.

The analyzed DSLs support a global storage using the *blackboard* behavioral design pattern [90], which is a key-value store. No scopes are supported, all keys are global. The blackboard is used for communicating, both within the model and with the rest of the system. The model and the system read and update the blackboard asynchronously.

1. For example, the PyTrees documentation states that the language provides a *good blend of purposeful planning towards goals with enough reactivity to shift in the presence of important events*; <https://py-trees.readthedocs.io/en/dev/background.html>

TABLE 3: Behavior tree concepts and corresponding language elements in BehaviorTree.CPP PyTrees_ros and PyTrees

concept	BehaviorTree.CPP	PyTrees PyTrees_ros
Simple Node	subclasses of ActionNode ConditionNode	behaviour.Behaviour
Composite	subclasses of ControlNode	classes in composites
Sequence	Sequence, SequenceStar ReactiveSequence	composites.Sequence
Selector	Fallback, FallbackStar ReactiveFallback	composites.Selector composites.Chooser
Decorator	subclasses of DecoratorNode	classes in decorators
Parallel	ParallelNode	composites.Parallel

Both BehaviorTree.CPP and PyTrees_ros offer the four basic categories of control-flow nodes and the two basic execution nodes. To illustrate the abstract syntax, we provide a meta-model that was reverse-engineered from BehaviorTree.CPP’s XML format in Fig. 6. These concepts are further detailed in Table 2.

Simple Nodes

Recall that simple nodes, or leaves in the syntax tree, are either conditions or actions. Actions realize the basic computation in the model. Users of our DSLs need to implement custom action nodes—classes obeying the Action interface that contain the Python or C++ code to be executed whenever a node is ticked. Conditions calculate a value of a Boolean predicate and convert it to a success or failure value.

BehaviorTree.CPP supports different types of synchronous and asynchronous action nodes. Meanwhile PyTrees_ros supports mainly synchronous type. The simplest action nodes are synchronous, so they terminate quickly and return success or failure immediately. Asynchronous nodes may also return a ‘running’ status and use some form of concurrency to continue operation. The execution engine will attempt to trigger them at the next epoch again.

Composite Nodes

Recall that composite nodes are internal nodes of a behavior tree. Their main function is to define the order of traversal at every time epoch (at every trigger). Unlike the simple nodes, which need to be implemented by the users, our DSLs provide a range of predefined composite nodes. The *root* node is the composite node that serves as an entry point for every traversal, it contains another node as the body. This node is re-entered to start every traversal.

The semantics of the different composite nodes in our studied DSLs follow the literature on behavior trees, which we described in Sect. 2. A *sequence* node is similar to a *forall* higher-order function, which is standard in many programming languages. A *selector* node is similar to *exist*. We confirm that in the implementations, *parallel* node is really a misnomer: It does not execute nodes concurrently, but it generalizes sequence and selector to a range of *policies*, described in each DSL’s documentation.

Since the execution is always a traversal of the entire tree, there is no direct support for jumps (*goto*). Instead, composite nodes can affect the traversal locally, in stark contrast to state machines. In the studied DSLs, a typical change of control allows an arbitrary change of state, often cross-cutting the syntax tree, depending on the returned status of a node. After each tick of composite nodes, and by propagation also simple nodes, explicit status is returned. Our DSLs support the same set of statuses as in the literature on behavior trees (see Sect. 2). These values propagate upwards during the tree traversal according to the semantics of composite nodes.

Decorators

Decorators are implemented as unary composite nodes (only one child) in the studied DSLs. They decorate the sub-trees and modify their data or control-flow. `BehaviorTree.CPP` and `PyTrees_ros` offer a wide range of constructs for Decorators. An *Inverter* flips the return status of a child between success and failure. A *Succeeder* always succeeds regardless the status returned by its child node. A *Repeat* node, which is stateful, acts like a for-loop: it continues to trigger the child for a given number of ticks. It increments an internal counter at every trigger. The node succeeds (and resets the counter) on hitting a set bound. It fails (and resets the counter) if the child fails. A *Retry* node resembles a repeat node. Its main goal is to make a flaky node succeed. Like Repeat it can run a node up to a set number of times, but unlike Repeat, it only retries when a node fails and it retries immediately without waiting for the next epoch. It fails if the child failed in a given number of attempts.

OBSERVATION 1. The concepts offered in behavior tree languages, and their semantics, stem from domain needs. The studied behavior-tree DSLs gather a number of constructs based on patterns that, according to users and developers, are frequently found in high-level control of autonomous systems.

Openness

The openness and indefiniteness of behavior trees are probably their most interesting aspects, after the time-triggered coroutine-based model of computation. Others have also noticed this in the context of variability in DSLs [91].

Unlike in *Ecore* [23] or UML, the languages' meta-models are *not fixed*. The DSLs provide the meta-classes for composite nodes, while it leaves the simple nodes abstract or only gives them bare bones functionality (cf. Fig. 6). A user of the languages is expected to first extend the meta-model by implementing the basic action nodes, then link them together in a syntax tree, possibly using an external XML file. This practice vaguely resembles stereotyping [47]. Obviously, a user of *Ecore* can extend the meta-model classes and give them new functionality at runtime as well. However, such use of *Ecore* is considered advanced and is seen rather rarely. The difference is that of degree: there is essentially no way to consider using behavior-tree DSLs without creating custom nodes.

Prerequisites (User Demographics)

The open nature of behavior-tree DSLs means that the experience of building and debugging models resembles very much language-oriented programming as practiced

in the modeling- and language-design research community. One constantly deals with meta-classes, composing them, traversing them, and so on. Anybody familiar with building DSLs on top of *Ecore* or similar frameworks will definitely experience a *déjà vu*, when using either `PyTrees_ros` or `BehaviorTree.CPP`.

Given that many robotics engineers, and many ROS users, lack formal training in computer science and software engineering [92], it is surprising to us that this design seems to be well received in the community. Even within software engineering, language implementation and meta-programming skills are often considered advanced. Yet, using behavior-tree DSLs requires such skills. A challenge for the modeling community is lurking here: to design a behavior-tree DSL that, while remaining flexible and easy to integrate with large and complex existing code bases, is much easier to use for a regular robotics programmer.

OBSERVATION 2. The flexibility and extensibility of the studied behavior-tree DSLs require language-oriented programming skills from developers. The software-language engineering community could contribute by designing an accessible, but still flexible, dialect of Behavior Trees.

Separation of Concerns

Behavior-tree DSLs support platform-specific models (PSMs) built as part of a specific robotics system to control behaviors at runtime. The models are used to simplify and conceptualize the description of behavior. The ability to reuse the same models with other hardware or similar systems is not (yet!) a primary concern. The studied behavior-tree DSLs not only are PSMs, but tend to be very tightly integrated with the system. Custom nodes tend to refer to system elements directly and interact with the system API. As a result, it is hard to use created models separately from the robot. While Groot can visualize a standalone XML file of a model, a working build environment of ROS is needed just to visualize the syntax of a `PyTrees_ros` model. This may mean not only an installation of suitable Python and ROS libraries, but, for example, a working simulation of the robot, or even the hardware environments. You need to launch the system and inject a visualization call to inspect the model!

It is in principle possible with both DSLs to build models that are completely decoupled from the system. It suffices to route all communication with the system via the blackboard. `BehaviorTree.CPP` provides dedicated XML primitives for this purpose, allowing the entire behavior to be programmed in XML, provided the rest of the system can read from and write to the blackboard. This separation allows models to be processed outside the system for visualization, testing, grafting into other systems, and so on. We definitely think this is a good architectural practice to follow. Nevertheless, it is not what we observed in real-world models (cf. Sect. 6). Most models mix the specification of behavior deeply with its implementation, making separation virtually impossible.

OBSERVATION 3. Behavior-tree models implemented with our DSLs tend to be deeply intertwined with behavioral glue code linking them to the underlying software system.

This makes operating on models outside the system difficult, hampering visualization, testing, and reuse.

4.2 Concepts and Semantics in State-Machine DSLs

In the following, we reflect on the syntax and semantics that we observed in the state-machine languages State MACHine (SMACH) [56] and Flexible Behavior Engine (FlexBe) [58].

The implementation of state machines in both DSLs is event-triggered, or reactive, which is similar to the traditional implementation of state machines in different domains. Compared to the studied behavior-tree DSLs, the state-machine DSLs do not have an explicit notion of ticks and of reoccurring traversals. States are only entered once at the beginning of the model execution, revisited only if the control-flow gets there as a result of reactions to external events. In FlexBe, the current state could be interrupted and a transition based on external events could be triggered, allowing reactive programming. The interruption is possible due to an autonomy-level threshold that is associated with a state to allow human-in-the-loop decision incorporation. In contrast, no reactivity is available in SMACH, and the current active state blocks the execution until an outcome is returned.

Both DSLs resemble Moore machines (action-on-state) [93] from the computational perspective. Users are expected to implement a state-based interface as a Python class, `State` in SMACH and `EventState` in FlexBe. Unlike the UML's state-machine implementation of Moore machines, there is no entry/exit actions associated to the state interface. There are different sets of provided functions, with the constructor `__init__` and the execution loop `execute` being the most important parts to implement the associated action to a state. The constructors in both DSLs require listing the outcomes to be returned by a state (triggering events). Other types of state interface exist in both languages, like `ConditionState` in SMACH and `LockableState` in FlexBe. More details can be found in their documentation [60], [61].

To deal with the data flow between states and the system, `UserData`, a locally scoped key-value dictionary is used. Compared to `blackboard`, `UserData` defines local input-output ports to access the needed data, while `blackboard` is more of a global storage accessed by any node. It has been highlighted by online discussion forums² and by researchers [68] that the current implementation of `blackboard` is causing name-clashes and unwanted overwrites when the behavior trees grow. Interestingly, by getting inspiration from the scoped input-output ports in state-machine languages (github.com/BehaviorTree/BehaviorTree.CPP/issues/41), developers of our behavior-tree DSLs added changes to scope `blackboard` (behaviortree.dev/migrationguide, py-trees.readthedocs.io/en/devel/changelog.html#x-2019-11-15-blackboards-v2).

SMACH and FlexBe support hierarchical nesting by means of the design-pattern container, so users can create hierarchical state machines. A container is simply a Python module that could be extended to support different execution semantics. This design pattern is also used in the studied languages to define constructs for

common high-level control-flow patterns. We distinguish between nesting container and control-flow containers according to their functionality. `StateMachine` in SMACH and `OperatableStateMachine` in FlexBe are for defining a state-machine model. Both are also used for nesting and creating hierarchical state machines.

In SMACH, `Concurrence`, `Sequence` and `Iterator` are some of the offered control-flow containers. `Sequence` acts similar to the behavior-tree control-flow node `Sequence`. The states are executed sequentially according to a predefined order. `Iterator` is a repeat-until loop that iterates over states until a specified outcome is reached. Finally, `Concurrence` allows multiple sub-states to be active at the same time through threading. Similar to the behavior-tree `Parallel` node, no parallelism of execution is available. An outcome policy defines the concurrency container outcome using a key-value dictionary (container outcomes are the keys and the sub-states' potential outcomes are the values). In FlexBe, only concurrency behavior is supported through `ConcurrenceContainer`. An example of SMACH iterator is provided in Fig. 7.

In addition to offering frequent control-flow patterns, FlexBe provides meta-classes for common states through the API `flexbe_state` and a separate state library `generic_flexbe_states`. A documentation of the states from `flexbe_state` is available in [58, app. A.1].

Providing constructs for frequent control-flow patterns seems common in the studied behavior-tree and state-machine DSLs. This might relate to the nature of robotic missions that tend to have a sequence of actions or some iterative tasks, enforcing the need for the behavior-modeling language to accommodate this type of behaviors. Implementing behavior design-patterns as language constructs in modeling languages is sometimes a problematic design decision, because it might increase language complexity. However, as stated by Bosch [94], it is actually the lack of expressive constructs that increase language complexity and user overhead. A language should fulfill its domain needs. In our previous study [28], we found a clear difference in the supported constructs in our comparison between the studied UML behavior-modeling languages (state diagram and activity diagram) and behavior-tree DSLs. A similar observation regarding the need to customize UML modeling languages was reported by Whittle et al. [95]. It is actually preferred in practice to use DSLs over UML modeling languages due to the expressiveness needed in robotics [24]. Thus, it is surprising to observe such a design decision by language developers, and it seems like a need for robotics applications.

OBSERVATION 4. Constructs for frequent control-flow patterns seem to be a common need in modeling robotics missions. Our studied state-machine and behavior-tree DSLs accommodate these needs using containers and composite node concepts, respectively.

Openness

Openness is a common feature in the studied languages due to the nature of robotic missions. Similar to behavior trees, the state-machine DSLs do not constrain users with fixed models and implementations. Users are provided with meta-classes for containers and states, and they can extend them. The

2. For example, the problem is raised in `BehaviorTree.CPP` GitHub issues page: #18 #41 #44

extension of control-flow types is possible, however we have not observed any such customization in the analyzed projects (see Sect. 6), just like the behavior-tree projects. Further investigation by including the users of languages would be required to determine the reason.

This design pragmatically supports openness of the language and makes adaptation to diverse scenarios in robotics easy. The openness seems to be required due to a lack of agreement in the robotics community about the ideal control model for robot behavior. Since this question is likely to remain open for a long time, the design allows users to adapt the language as they see fit when building robots.

5 LANGUAGE IMPLEMENTATION (RQ2)

This section focuses on analyzing the behavior-tree libraries in the first three rows in Table 1, and the state-machine libraries in the first two rows in the second half of the table, all set in bold font. For the identified libraries, we broaden the scope of our analysis by inspecting their implementation techniques and practices.

5.1 Behavior-Tree DSLs: Language Design

Turning our attention to how behavior-tree languages are implemented from the language-design perspective, the first striking observation is that both languages are predominantly distributed as libraries, not as language tool chains, or modeling environments. `BehaviorTree.CPP` is implemented as a C++ library, packaged as a ROS component, easy to integrate with a ROS-based codebase [67]. In contrast, `PyTrees` is a pure Python library. It has an extension `PyTrees_ros` which packages `PyTrees` as a ROS package and adds ROS-specific nodes.

Syntax and Visualization

Both DSLs come with ways to visualize models as graphs. `BehaviorTree.CPP` even has a graphical editor and a visual runtime monitor for its models called Groot (which created the behavior tree in Fig. 1).

Nevertheless, it is important to understand that behavior trees are *not* a visual modeling language in a traditional sense. First, in both libraries, the models are constructed in a text editor, in a mixture of C++, respectively Python. Second, the models are constructed directly in abstract syntax, by instantiating and wiring abstract syntax types. For convenience, and to support Groot, `BehaviorTree.CPP` offers an XML format, which can be used to write the tree syntax in static files. Listing 1 shows the XML file for the mission displayed in Figure 1. This file is interpreted at runtime, and an abstract syntax tree is constructed from it dynamically. Third, crucially, the types of nodes (and, thus, the XML file in `BehaviorTree.CPP`) do not constitute the entire meaning of the model. An important part of the model is embedded in C++/Python code that is placed in the methods of the custom node classes. This part of the model is neither modifiable nor presentable in the graphical tools.

Finally, recall that `BehaviorTree.CPP` is realized as an external DSL through Groot and the XML-like format, while `PyTrees_ros` constitutes an internal DSL, since it does not have similar tools. From our experience analyzing their models (cf. Sect. 6), we can confirm that the `BehaviorTree.CPP`

models are much easier to comprehend, and the availability of its visual editor Groot has made it faster to analyze the behavior tree models than `PyTrees_ros` models.

Listing 1: The same example in Figure 1 shown in `BehaviorTree.CPP` XML notation.

```

1 <root main_tree_to_execute="MainTree" >
2   <BehaviorTree ID="MainTree">
3     <SequenceStar name="MainSeq">
4       <Action ID="CollectWaypoints"/>
5       <RetryUntilSuccessful num_attempts="10" >
6         <Negation>
7           <Sequence name="ExplorationSeq">
8             <Action ID="PopWaypoint"/>
9             <Action ID="MoveBase"/>
10            <Action ID="Explore"/>
11          </Sequence>
12        </Negation>
13      </RetryUntilSuccessful>
14    </SequenceStar>
15  </BehaviorTree>
16 </root>

```

Concurrency

The design of our behavior-tree DSLs does not prescribe the model of concurrency, and implementations vary. For instance, the `BehaviorTree.CPP` engine uses a single-threaded execution engine. All nodes are supposed to return a status immediately. If an operation lasts longer, it should spawn an asynchronous logic and inform the behavior-tree engine that it continues to run (or that it completed, if ticked again upon the completion). So, the programmer has complete freedom to choose the concurrency mechanism.

OBSERVATION 5. The implementations of our behavior-tree DSLs support both interleaving and true concurrency indirectly by resorting to the underlying ROS platform. The model of concurrency is not defined strictly in the language, but is, instead, left largely to the users.

An internal or external DSL?

Our DSLs are unusually open. `BehaviorTree.CPP` is technically an external DSL, but its implementation exposes aspects of dynamic internal DSLs. The programmer can both create models in XML (external, static), and create new node types or modify the shape of the syntax tree at runtime (dynamic). `PyTrees_ros` is an entirely dynamic DSL, where new node types and Python code can be freely mixed, like in internal DSLs.

An interpreter or a compiler?

Our DSLs' models are interpreted. Once the abstract syntax tree is constructed, the user is supposed to call a method to trigger the model once, or to trigger it continuously at a fixed frequency. This does not seem to depart far from other applications of models-at-runtime [29], [30]. `BehaviorTree.CPP` uses template metaprogramming instead of code-generation, which allows to offer a bit of type-safety when implementing custom tree nodes, without exposing users to any specialized code-generation tools. Using the library appears like using a regular C++ library. As expected, no static type safety is offered in `PyTrees_ros`.

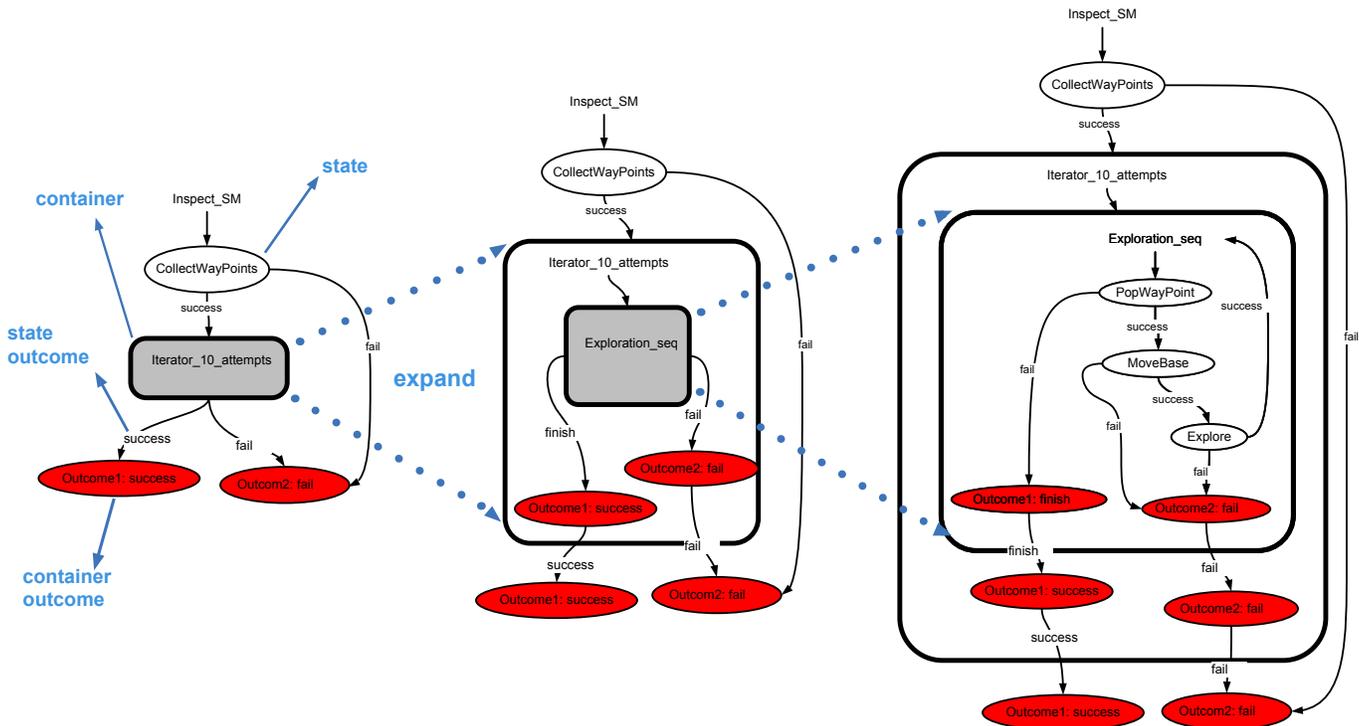


Fig. 7: A state-machine example representing the mission in Fig. 1 and Fig. 3, using `smach_viewer` syntax. On the left, a top view is provided. The middle and right illustrations represent expanded iterator and nesting containers, respectively

5.2 State-Machine DSLs: Language Design

SMACH and FlexBe are open-source software frameworks written in Python for building and monitoring state machines and hierarchical state machines. Similar to `PyTrees_ros`, SMACH can be used without the ROS system and it has a ROS binding through `smach_ros`.

Looking at how these state-machine DSLs are implemented from a language-design perspective, there is a difference in how SMACH and FlexBe are realized. Although FlexBe can be used directly as a Python library, it is used as a modeling environment, since it offers a graphical user interface (GUI) called FlexBE App (github.com/FlexBE/flexbe_app). It is used to construct state machines using a graphical notation, and to monitor and modify the state machine during runtime. On the other hand, SMACH is mainly used as a pure Python library, and it has a package `smach_ros` to integrate it with ROS. It has a GUI called `smach_viewer` (wiki.ros.org/smach_viewer) for inspecting and debugging an already created state machine, but not constructing or modifying it. Although SMACH is ROS independent, its viewer cannot be executed without having a running ROS.

Syntax and Visualization

SMACH and FlexBe have their own notation, different from the UML standard in Fig. 4. SMACH uses an ellipse shape for states and a double-frame ellipse shape for nested machines, and FlexBe uses rectangle shapes. FlexBe uses a dedicated notation for initial and final states like UML, while SMACH does not. Figure 7 shows a state-machine example using the `smach_viewer` syntax. Events are called outcomes in SMACH

and FlexBe. SMACH distinguishes between a container outcome and a state outcome in the visual syntax. The outcome of a state that caused a transition (triggering event) is represented on the transitioning arrow. The outcome of a container, like the `Iterator_10_attempts` in Fig. 7, is represented explicitly as a red ellipse, which looks like a state. This deviates from the typical state-machine syntax.

The two languages are used differently for constructing the state-machine model. FlexBe mixes graphical and textual syntax. The graphical syntax is constructed using the FlexBE App (FlexBe graphical GUI) in a drag-and-drop manner, then an interpreter generates the textual syntax as Python code. The generated code represents the model structure, and certain parts of it can be edited to provide flexibility to developers. For each state, a stub is auto-generated for class methods consisting of basic functions, e.g., a constructor, execution function, and so on. Developers are expected to complete the auto-generated stubs for states in Python code. Meanwhile, SMACH users construct the model's abstract syntax directly in a text editor in Python.

FlexBe uses MDE techniques in the construction of the model. It makes use of model-based design processes through (semi-)automatic generation of code from the models. Syntactic and semantic checks for model consistency are also supported. Consequently, separation-of-roles is supported in FlexBe. `BehaviorTree.CPP` also supports separation-of-roles [67], in addition to separation-of-concerns, by providing a GUI for constructing the graphical model and generating the textual syntax in an XML format from the model, which is useful, but the language still requires manual repetitive coding. While FlexBe reduces the manual coding by generating boilerplate code, such as the import of needed

libraries and default functions for the state initialization. If adopted by behavior-tree languages, this functionality might reduce programming effort and reduce syntactic errors. However, since custom-execution functions still need some manual coding, it is not fully applicable without some programming background.

OBSERVATION 6. One of the state-machine DSLs has used concepts from model-driven engineering that reduce repetitive coding and code syntactic errors. Enhancing the user support of behavior-tree DSLs for similar features could reduce programming effort and syntactic errors.

Concurrency

Similar to behavior trees, the model of concurrency is not strictly enforced in the implementation. The user can exploit any concurrency semantics available in the ROS platform. In both libraries, every state-machine container is executed in its own thread, and its *UserData*, the locally scoped dictionary of key-value pairs, is not shared between the different state-machine containers. To facilitate data sharing between two running state machines, one needs to resort to external synchronization and communication mechanisms.

The analyzed DSLs offer a concurrency container. In both DSLs, Python threading is used allowing multiple states to be active at the same time, each running at its own rate. States are activated sequentially, not parallel due to using the Python threading (interleaving).

An internal or external DSL?

The openness of `FlexBE` is similar to other modeling languages using model-based design. Similar to `BehaviorTree.CPP`, it is realized as an external DSL. State-machine models can be created using the graphical notation in the `FlexBE App`. Syntactic and minor semantic verifications run before generating code to ensure model consistency, such as the existence and correctness of the associated *UserData* with each state. Extension of the model and states is possible offline. Once the model and states are instantiated, runtime modification is limited to the structure of the state-machine model, e.g., add and remove states, and transition function, while modifying the implementation of a state is not possible. On the other hand, `SMACH` is similar to `PyTrees_ros` and it is realized as an internal DSL, where extending the model and states is done easily by simply editing the Python code.

An interpreter or a compiler?

Both libraries execute the state-machine model through interpretation at runtime. `FlexBE` allows modifications at runtime. In `FlexBE`, modification to a running state machine is simply realized through patches—a standard for minimal summaries of the modifications. Patches are applied by taking the difference between old and new code and only re-importing the changes to the source code. In order to apply the modifications, the modified state is locked to prevent transitions during the modification using the Python threading lock (docs.python.org/3/library/threading.html), then the Python module is reloaded (using the built-in `reload()` function). No modification is applied before the `FlexBE` engine has run verification checks to ensure consistency (cf. Sect. 5.2). So, overall, the `FlexBE` developers implemented

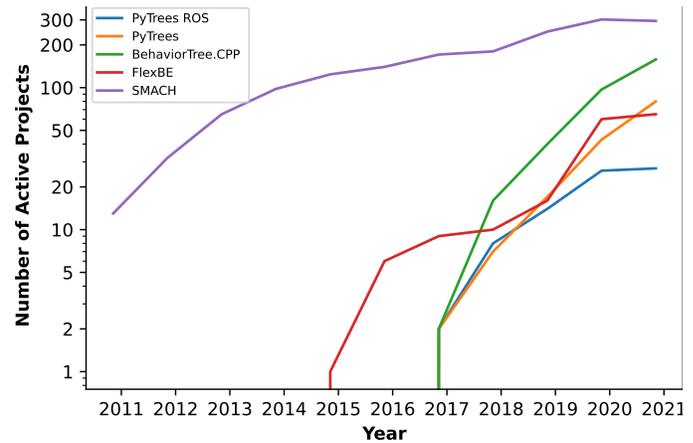


Fig. 8: Usage of the state-machine and behavior-tree DSLs in open-source projects on GitHub over time

a pragmatic state-machine swap at runtime, largely relying on Python’s reload support. `SMACH` has no verification step for the created model before execution.

6 BEHAVIOR-TREE AND STATE-MACHINE MODELS (RQ3)

Our mining returned hundreds of GitHub projects per DSL. In the following, we report on our exploratory investigation of the usage of state-machine and behavior-tree DSLs in these projects. First, we present an overview of the different languages’ popularity among the mined open-source projects. Second, we provide our analysis results for a sample of these projects. Our analysis covers the structure of the sampled models and their code reuse among the projects.

6.1 Language Popularity

The initial mining returned 1,086 projects using our state-machine DSLs and 271 projects using our behavior-tree DSLs (see Sect. 3.3 for the search terms). To visualize the language (i.e., library) use in open-source projects over time, we tracked the number of active projects per year per library in these mined projects and plotted each library’s trend line.

Figure 8 shows the trend. Until 2015, `SMACH` was the only available language among our five DSLs, and its usage has been increasing steadily over the past five years (1.2 times more each year). In 2015, the other four languages were released. However, their usage only started to increase significantly in 2018. In general, the use of the languages has grown over the past five years. In 2021, `BehaviorTree.CPP`’s and `PyTrees`’s usage in open-source projects reached almost ten times what they were in 2018. We hypothesize that this significant increase in the last three years is due to ROS adopting it as the core component for its navigation stack Navigation 2 [36]. For `PyTrees`, it could be because it was the only Python implementation for behavior trees offering a stable and actively maintained language and a wide range of constructs for node types. Although the `FlexBE` usage in open-source projects is lower than `SMACH` (78% lower than `SMACH` in 2021), its overall usage growth since 2018 is much higher than that of `SMACH`. In 2021, the `FlexBE` usage was 6.5 times higher than in 2018, while that of `SMACH` only increased around 1.6 times.

TABLE 5: Projects and models extracted per library

	SMACH	FlexBe	Behavior-Tree.CPP	PyTrees_ros
projects	560	60	141	28
models	2065	442	595	63

6.2 Characteristics of the Models

After applying the multiple filtering steps in our mining, we ended up with 620 projects that include 2,507 state-machine models used in them, and 169 projects that include 658 behavior-tree models. Table 5 shows the number of projects and models per DSL. In a project, a file with a full definition of a model is counted as a model, resulting in projects with multiple models. We randomly sampled 75 models per language. We analyzed a total of 150 models of behavior trees and state machines (75 models each) belonging to 43 projects (25 behavior trees and 18 state machines), as summarized in Table 4. The difference in the number of projects stems from the choice to sample the same number of models rather than projects.

In our sample dataset, we found eleven different domains, as shown in Figure 9. The dominating domains are robotic patrolling, waiter robotics and pick&place applications. We identified three types of projects: (1) research projects belonging to labs, research groups, or competing teams in events, (2) company projects belonging to companies producing robotic solutions, and (3) unknown projects, for which we could not find enough information to characterize. Table 4 presents the types of our projects with a link to their corresponding organization and their GitHub repository by hovering over the blue text.

We noticed a variation between the structural properties of behavior-tree and state-machine models in the analyzed sample. As a start, state-machine models have a right-skewed model-size distribution with an average model size of 9 across the population. Similarly, behavior-tree models have a right-skewed model-size distribution, but the average model size is three times larger than that of state machines (the average model size is 26 across the population). Interestingly, the average branching factor BT.ABF is 3, which is small, meaning the developers kept the trees in a somehow manageable size. The reason behavior-tree models seem to have a larger model size compared to state machines could be linked to the former representing both execution nodes and control-flow nodes, which state-machine models lack [16]. Another reason could be linked to state-machine models becoming complex once their size is big, affecting their understandability [96], [97]. Consequently, developers might try to keep them small. For behavior trees, there is no work in the literature that confirms the optimal size of models. However, through our analysis of models we noticed that the combination of moderate size and branching of nodes makes a model easy to navigate and understand. By moderate, we refer to the average reported values for both BT.size and BT.ABF.

Another structural property we were interested to capture is nesting and depth. According to existing empirical studies [98], [99] the nesting level affects the understandability of state-machine models, so shallow models should be preferred.

In our sample, state-machine models have an average nesting level of one, and behavior-tree models have an average depth of five (similar across the different DSLs). In state-machine models, those using FlexBe implemented hierarchical state machines, where the nesting level is above one, more often than models using SMACH (11 out of the 14 hierarchical state-machine models). It appears that the developers in our sample projects kept the models shallow, both when building state-machine and behavior-tree models. Unfortunately, there is yet no study on the impact of a behavior-tree’s depth on its complexity that could justify the developers’ tendency to design shallow models. A justification could link the design decision to understandability, which would be similar to state-machine models. However, a proper investigation is required to confirm.

Now we turn to the use of different language constructs in the sampled models, starting with behavior trees. In our behavior-tree sample, 66% of the nodes in the dataset are leaf nodes (1,228 out of 1,850 nodes), and 34% are composite nodes. Table 6 summarizes the usage of composite nodes for each library and for the total population of behavior-tree models.

Most of the composite nodes in our projects are of type Sequence (56% of total composite nodes) followed by Selector type (21% of total composite nodes). The Parallel node concept, which generalizes Sequence and Selector, was not used much, only for 7% of all composite nodes. This might explain why standard libraries for programming languages normally do not include generalizations of existential and universal quantifier functions (exists and forall)—these use cases seem to be rare. The re-entrant nature of the behavior-tree DSLs allows to use Parallel to wait until a minimum number of sub-trees succeed. This, however, does not seem to be used as often as we expected.

Decorators are used relatively rarely in PyTrees_ros models, where they constitute 6% of the composite nodes. This is likely explained by the fact that it is easier to apply the transforming operations directly in the Python code, using Python syntax, than elevating it to behavior-tree abstract-syntax constructors. The situation is different with BehaviorTree.CPP, where decorators are used almost three times as often (19% of composite nodes). Here, the benefit of using the decorators (data-flow operators) of the behavior tree instead of C++ allows them to be visualized and monitored in the graphical editor (Groot). No such tool is available for PyTrees, so likely larger parts of the model may “leak” to the code. This demonstrates that users of behavior trees often have a choice of what is in scope and what is out of scope for a model. By model scope, we refer to the boundaries of a model which decide what is included in the model and what is left to be programmed outside of the model. This is a property that clearly distinguishes general-purpose languages (GPLs), such as Python and C++, from DSLs. Yet, in our experience, the competence of deciding the model scope and the precision level is only rarely (with exceptions [23]) discussed in the teaching and research literature.

Finally, we observed that none of the models implement their own custom nodes. They rely on the extensibility of behavior trees using new custom operators (decorators). The available off-the-shelf decorators in BehaviorTree.CPP and PyTrees_ros were sufficient to create a custom behav-

TABLE 6: Usage of different behavior-tree composite nodes to the total of them per library and in total for all models

library	composite nodes			
	Sequence	Selector	Decorator	Parallel
BehaviorTree.CPP	57%	19%	19%	6%
PyTrees_ros	53%	28%	6%	13%
share in population of all models	56%	21%	16%	7%

ior to change an action/condition status, or to customize an action duration, e.g., wanting to execute an action without waiting, retrying an action n times before given up, or repeating an action n times.

Going back to Fig. 1, the decorator (`RetryUntilSuccessful`) was used to create a conditional loop that executes the sub-tree under (`ExplorationSeq`) 10 times, unless the task fails, which is inverted into a success by an (`Inverter`). The developers were able to model this without having to use a while-loop or a similar general control-flow structure in the script.

OBSERVATION 7. The studied behavior-tree DSLs offer a range of concepts that are well suited to roboticists, but the usage of the offered concepts differs according to the GUI support of the languages.

For state-machine DSLs, we focused on the control-flow containers as a concept equivalent to the composite nodes in behavior trees. As mentioned in Sect. 4.2, `SMACH` offers a range of control-flow containers, sharing only Concurrency behavior with `FlexBe`. For this reason Sequence and Iterator data is reported only for `SMACH`. In general, control-flow constructs in our sampled state-machine models are used far less than in behavior trees. Only 12% of the state-machine models used some kind of control-flow construct, while in the behavior-tree models all of them used some type of control-flow node. Concurrency is used in 11% of all models, with models using `FlexBe` being responsible for the majority of this number (7 out of the 8 models used `FlexBe`). Although, Concurrency is used less in `SMACH` than in `FlexBe`, it makes for 67% of the total control-flow constructs used in `SMACH` projects. Iterator is the second most used control-flow construct in the projects using `SMACH` with 33% of the total control-flow types. Sequence is never used in our sample.

It is not clear why the offered control-flow constructs are not as popular in our analyzed sample of state-machine models compared to the behavior-tree models. It might be related to the syntax of state-machine models, or users not being familiar with them. By syntax we mean that state machines have a structure that is easy to understand, where states represent the status of a system and transitions facilitate the control-flow between states. Adding a layer of complexity to the model by using the control-flow constructs offered by the DSLs is less needed. Also, it might be a similar case to the use of `PyTrees_ros`'s Decorators, where it is easier to apply the pattern directly in Python syntax, than elevating it to the state-machine abstract syntax.

TABLE 7: Frequency of reuse patterns per library for skill and task level

library	intra-model referencing		clone-and-own		inter-model referencing	
	skills	tasks	skills	tasks	skills	tasks
<code>SMACH</code>	-	-	20%	13%	35%	8%
<code>FlexBe</code>	1.3%	-	-	16%	32%	-
total for state machines	1.3%	-	20%	29%	67%	8%
<code>Behavior-Tree.CPP</code>	25%	13%	3%	39%	37%	3%
<code>PyTrees_ros</code>	5%	5%	13%	9%	19%	-
total for behavior trees	30%	18%	16%	48%	56%	3%

6.3 Reuse

After presenting the core structural characteristics of our models in the sample, we now shift to reuse as one of the major issues in robotics engineering software [1], [6], [80], [81], [82], [100] and control architectures [5], [32]. To facilitate reuse, a decomposition of a robotic mission or behavior into modular components should be supported by the modeling language. A behavior-modeling language supporting modular design—consequently, reusable components—is an important aspect for overcoming challenges of robotics control architectures and enhancing robotics software maintainability and quality [5], [6], [32], [100].

In our work, reuse refers to the ability to use already implemented skill code (also known as action in other contexts), or reusing code of a repeated task (composed of different skills) in the same model or across models in a project. We use the terms skill-level and task-level in the remainder to reference each. For skill-level and task-level code reuse, we observed three patterns of reuse in our sample: *intra-model referencing*, *reuse by clone-and-own* [101] and *inter-model referencing*.

Intra-model referencing was mostly used by behavior tree models. On the task-level, it is implemented by creating a sub-tree for a repeated activity, then re-using it by reference in multiple branches in the model after passing new values to its parameters (usually by writing a new value to a blackboard). A skill-level implementation defines a leaf node as a function in the main model execution file, then reuses it by reference after passing new values to its parameters.

The behavior-tree models often exploited *intra-model referencing* on skill-level (30% of the behavior-tree models), while on task-level it was only used in 18% of the models. This pattern was rarely used in state-machine models. Only one reused its skills by reference, and few reused on the task-level. Figure 10 shows an excerpt from one of our behavior-tree models, presenting the different tasks for a robot in a retirement home. The red box highlights an example of intra-model referencing, where the developer wrapped the moving activity in the sub-tree (`Recharge`) and reused it in multiple parts of the model. An example on the skill-level is shown in the action `moveRobotPosition`, which was used in multiple parts in the model, only changing the values of parameters (name, approachRadius, x, and y).

Reuse by clone-and-own was the most used pattern for task-level reuse in the DSLs. In projects with multiple behavior-

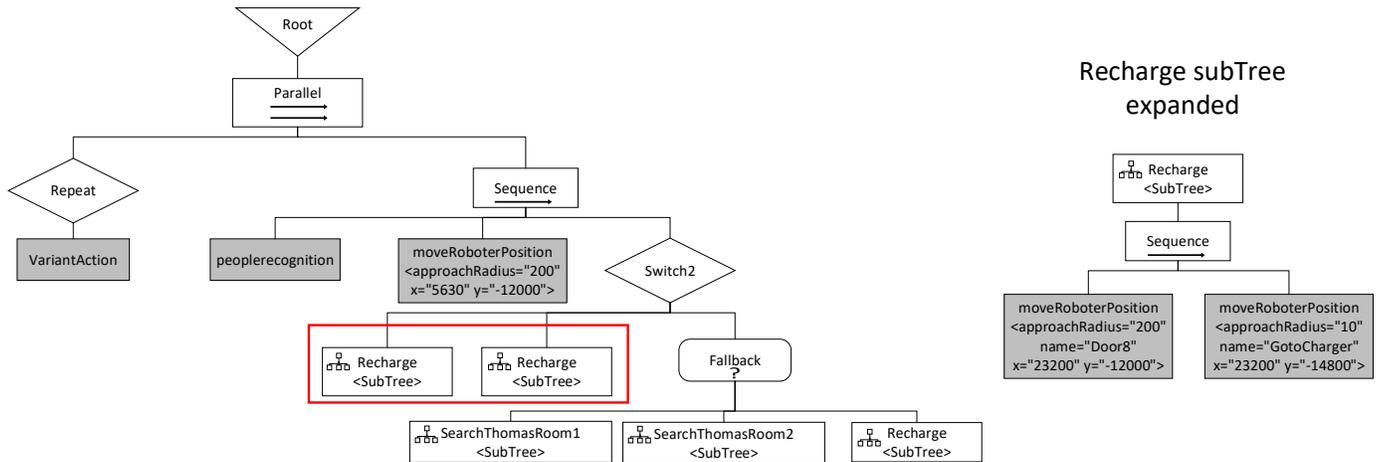


Fig. 10: Behavior tree model of a retirement home robot from project bundles. The red box highlights an example of a intra-model referencing for a sub-tree Recharge (expanded on the right side). A legend is shown in Fig. 2.

tree models, we observed that, when two behavior trees have the same activities, the similar parts (a sub-tree or the entire model) are reused after some minor changes, such as adding new nodes or removing old ones. Similarly in state machines, minor modifications are introduced to a model by adding or removing states to accommodate similar tasks. On the skill-level, clone-and-own was mainly implemented by minor modifications to the skill code.

Users of behavior trees and state machines favored using clone-and-own for task-level reuse in our sample. It was the most used pattern for task-level reuse compared to the other two patterns (48% of models in behavior trees and 29% of models in state machines used it). Clone-and-own was not as popular for skill-level reuse as task-level reuse. It was the least-used pattern in behavior-tree models (16% of the behavior-tree models) and second used pattern in state-machine models with only 20% of models reusing skills using this pattern. The Dyno project in Fig. 11, a drone-based parcel delivery project, includes two behavior-tree models: one for a parcel delivery mission (M1) and another one for a route scheduler mission (M2). These models are an example of task-level clone-and-own, where the developer reused the entire behavior-tree model for two different missions that share similar activities after proper modification depending on the mission.³

Inter-model referencing was the most used pattern for skill-level reuse in the DSLs. Repeated skills were implemented as action nodes or associated with states in external files that were imported later in the main model execution file. This allows projects with multiple models having similar skills to reuse these skills in different parts of the model. Similar implementation paradigm was used for task-level by defining a repeated task as a behavior-tree or state-machine model in an external file then it is imported and invoked in the main model execution file. The repeated task is plugged as a sub-tree in the main behavior tree or as nested state machine in the main state-machine model.

Inter-model referencing was the most used pattern for skill-level reuse among the different languages (used in 67%

of the state-machine models and 56% of the behavior-tree models). For task-level, only two projects from the state-machine dataset used it and one project from the behavior-tree dataset (corresponding to 8% of the state-machine models and 3% of the behavior-tree models).

An interesting observation was that inter-model referencing through predefined skills offered by FlexBe developers as either an API `flexbe_state` (github.com/FlexBE/generic_flexbe_states), or as separate state libraries `generic_flexbe_states` (github.com/FlexBE/generic_flexbe_states) and Flexible Navigation (github.com/FlexBE/flexible_navigation/tree/ros2-devel-alpha/flex_nav_flexbe_states). The skills were used in 73% of the FlexBe models. Similar behavior was observed across projects of SMACH and PyTrees_ros libraries by reusing ROS Navigation Stack skills, such as `move_base` package (in 55% of the SMACH projects and 60% of the PyTrees_ros projects). The data shows that predefined skills APIs could facilitate reuse, and could be worth considering by developers to facilitate reuse.

In general, we observed in our sample that the choice of reuse mechanism is influenced by the implementation of the DSL, and whether one reuses skills or tasks. It was noticeable that inter-model referencing was favored for skill-level reuse among the different DSLs, since defining a skill in an external file and importing it is a pretty straight-forward mechanism. For task-level reuse, inter-model referencing was the least-used method, while clone-and-own was more favored. Similar missions might share a set of skills, but the combination of them to form a task changes to accommodate specific needs, thus clone-and-own is the most used pattern for task-level reuse. Zooming into reuse for each DSL, projects using `BehaviorTree.CPP` have reported higher reuse frequency compared to projects using `PyTrees_ros`. These numbers can be related to `BehaviorTree.CPP` having a dedicated XML format to express the behavior-tree model and Groot for visualizing the model, instead of intertwining the model with the code like in `PyTrees_ros`. Visualization and abstraction of the model could be an aspect to explore for pushing toward reusing in robotic missions.

We conjecture that the identified simple reuse

3. The model can be found in full-size in the online appendix, [31] in addition to the models of the other projects.

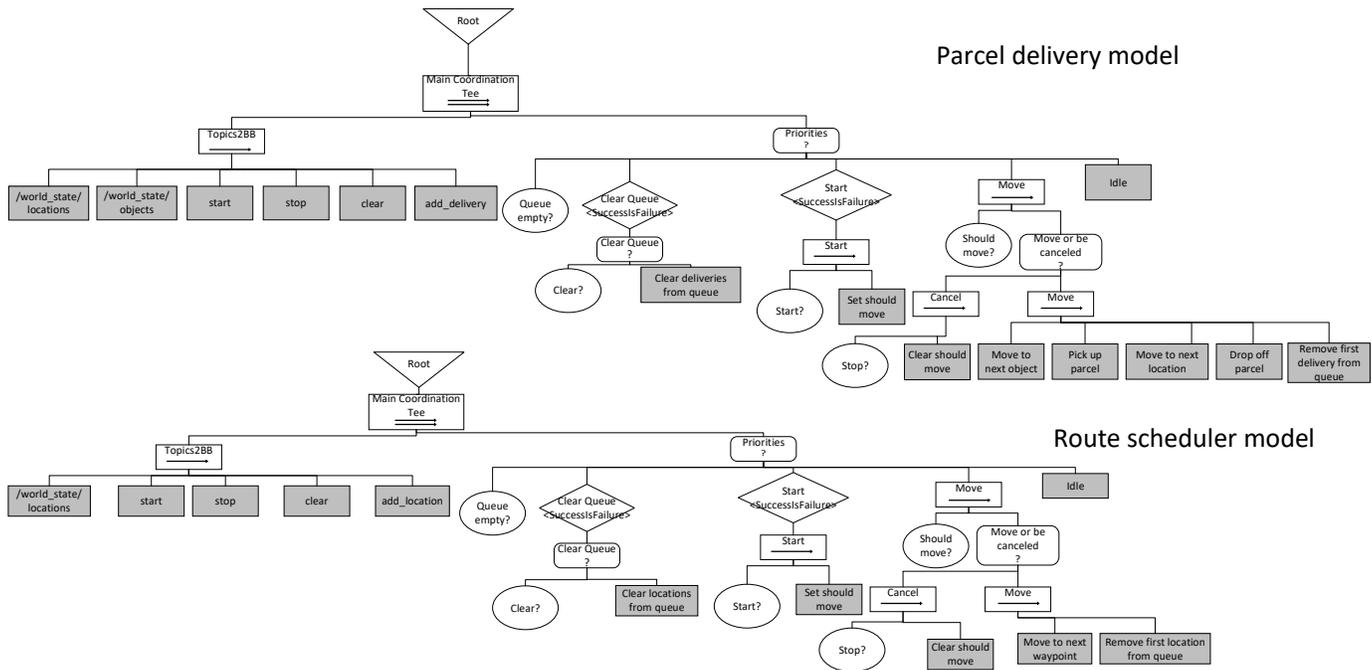


Fig. 11: An example of clone-and-own referencing in Behavior trees from project Dyno. Each model belong to a different mission (M1) parcel delivery, and (M2) a route scheduler. Legend in Figure 2.

mechanisms suffice for the identified robotics projects. It is less clear whether it would be useful to have safer and richer reuse mechanisms known from mainstream programming languages, including name-spacing and safe reuse contracts (interfaces), as they tend to be heavyweight for users to learn and use. More research is needed to determine whether sufficiently lightweight and safe reuse mechanism could be realized.

7 THREATS TO VALIDITY

Internal Validity. We provide a dataset of behavior-tree and state-machine models in robotics. Our dataset could include projects from courses or tutorials, and we could have missed projects with behavior-tree or state-machine models expressed in our libraries. To mitigate that, we applied a multi-step filtration mechanism that was adopted to our observations during the mining process. We derived our filtration steps from our data observations. After each step, we checked randomly a sample of the results to see if we needed to adopt the filtration mechanism.

Another threat that could affect the results, is possible errors in our Python scripts calculating the model metrics. In behavior trees, as a form of quality check, we manually counted node types and checked the script results against them. We excluded commented parts and unused node types in the behavior-tree codes. In state machines, we rechecked the automatic counting results by going through the models.

External Validity. A threat is the generalizability of our quantitative results and the identified reuse patterns, since we used a random sample of the mined models. We mitigated that by aggregating the models into pools of sizes according to their data distribution, then randomly sampling our dataset. This way, we made sure to capture different projects

with different model sizes. In addition, our project domains, presented in Fig. 9, show that our random sample covers different domain categories.

The list of identified open-source robotics projects might be missing examples from Bitbucket and GitLab. Both platforms are used in the robotics community; however, they do not provide a code search API, which makes it difficult to conduct a code-level search. We conducted a less precise query in Bitbucket and GitLab using *behavior trees* and *state machine* as search terms in the web interface. However, for behavior trees, we could not identify any real robotics projects from that search. For state machines, GitLab returned results for which it was hard to check if any of the subject libraries were used. We favored using GitHub for the flexibility of the provided APIs, for their integration with Python and other languages for mining, and because it is a platform widely used by developers to publish open-source projects.

We have only considered projects using Python and C++ libraries with ROS support, while there might be other open-source robotics projects out there. We acknowledge that limiting our search to ROS-supported languages might have resulted in missing other robotics projects. However, we focused on the two dominant languages in ROS, assuming that this is the most representative framework for open-source robotics.

8 RELATED WORK

In the literature, researchers compare—theoretically—behavior trees to popular control architectures in robotics, such as finite state machines, the subsumption architecture, and decision trees. They show that behavior trees generalize them [7], [16], [32]. They also discuss advantages and disadvantages of each control architecture based on design properties that are important for robotic control architectures.

Colledanchise et al. [15] describe the drawbacks of using state machines for multi-robot scenarios in comparison to behavior trees using illustrative examples. In another work, Colledanchise et al. [68] illustrate how to represent different kinds of behaviors from state machines in behavior trees, provide available software libraries and show what kind of representations they support. Modularity, reactivity, expressiveness, and readability are some of the properties of behavior trees that are studied in a formal-theoretical manner and compared to state machines by Biggar et al. [12]. Colledanchise and Natale [68] highlight that behavior-tree tools are less mature than tools available for state machines; however, their findings do not build on an analysis of available tools. In our work, we observed good language-design practices for the studied behavior-tree and state-machine DSLs (tools), and some sub-optimal ones. However, we cannot conclude that the studied state-machine DSLs were better designed than the behavior-tree DSLs. Each language design had its own advantages and disadvantages.

The related work only bases its findings to a very limited extent on software engineering practice in real-world robotics projects. In this work, we scope the analysis to behavior-tree DSLs in robotics from the software-language perspective, and we compare them to state-machine DSLs, which has not been done so far in the literature. Compared to previous work, we shift the comparison of behavior trees and state machines from the theoretical perspective to the real world. We do not report on the advantages, or disadvantages, of one model over the other. We report on the similarities and differences between the supporting languages of both models in robotics. We report on the state-of-practice by analyzing the usage of behavior-tree and state-machine DSLs in open-source projects. In our work, we focus on the structural properties and reuse-of-code in open-source projects, using the identified languages. Consequently, our work complements the related work. In addition, we provide behavior-tree and state-machine models as a community dataset, which has not been done so far. The dataset can be used for further research.

9 CONCLUSION

We presented a study of behavior-tree and state-machine DSLs, and of their use in open-source ROS robotics applications. We systematically compared the concepts available in popular behavior-tree DSLs and contrasted them with well-established DSLs for reactive modeling—the language of state machines. We mined open-source projects from code repositories and extracted the behavior-tree and state-machine models from their codebases. We analyzed the structure of a sample of these models, as well as how they use the concepts and how they reuse model code. We contribute a dataset of models in an online appendix [31], together with scripts and additional data.

Results and Implications

The paper displays popular DSLs designed *outside* of the language-engineering community for the vibrant domain of robotics. We believe that studying modeling and language-engineering practices is beneficial for both communities, as

it helps to improve language-engineering methods and tools, as well as to improve the actual practices and languages.

Our analysis showed that both groups of DSLs follow good design practices and even obtain some improvement by learning from each other. As a start, having a GUI that provides constructing, editing, and runtime monitoring of models is a good practice. A GUI supports visualization and building abstractions. Both `FlexBe` and `BehaviorTree.CPP` provide similar tools, which is reflected in their users reusing code more often compared to `SMACH` and `PyTrees_ros`.

In addition, our results illustrate that many of the modeling and language-engineering methods are relevant in practice. Developing DSLs in a rather pragmatic way, without hundreds of pages of specification documents and with a basic, but extensible meta-model, or even without an explicitly defined meta-model seems to be successful. Such a strategy seems to attract practitioners not trained in language and modeling technology, allowing practitioners who come from lower-level programming paradigms to raise the level of abstraction and effectively implement missions of robots in higher-level representations.

Concepts from software engineering are adopted in the design of the studied languages. `BehaviorTree.CPP` is built with separation-of-concerns and separation-of-roles in mind [67], making it possible to decouple models from the robotic system for visualization and testing. During model analysis, we managed to load models easily in Groot without the need to setup ROS or fulfill other requirements of the robotic projects. Also, inspecting the XML representation of the models was relatively easy. These are two features that we appreciated during the analysis. Unfortunately the state-machine DSLs and `PyTrees_ros` lack an easily accessible visualization tools, which required us to build custom ways to inspect models.

Not all the studied languages support the separation-of-roles to the same extent. `FlexBe` uses code generation from model-based design for boilerplate code, while `BehaviorTree.CPP` requires custom code in an internal DSL, which requires programming skills. Although code-generation, and external DSLs, are more constraining than the flexibility offered by programming in a general-purpose language with an internal DSL, support from code generators for creating repetitive parts seems valuable, as more aspects of modeling can be performed by domain experts. Furthermore, syntactic and semantic checks of model consistency are another natural feature of external DSLs exploited in `FlexBe` (these can be realized in internal DSLs as well, albeit at a higher cost). Still, the users of all the studied languages need to be familiar with language-oriented programming (meta-programming). This is despite the fact that most robot developers are not trained in this field. This poses an interesting challenge for the language design community to come up with designs that would reduce the friction of integrating models with the rest of the robot architecture.

Promoting and adopting mode-driven engineering practices has been on the rise in the robotics community to improve the reusability and maintainability of systems [6], [24], [79], [80], [100], [102], [103], [104], [105]. Through our analysis, we observed common practices across the different languages that are adopted from model-driven engineering specifically, and from software engineering generally,

which seem well received by practitioners. In addition, improvements to these behavior-modeling languages could be adopted and evaluated in the robotics community.

Future Work

In the future, we would like to build on top of current observations and conduct studies with users in the loop. Our goal would be to compare the usability, comprehension and expressiveness of the studied behavior-modeling languages, and the needed improvements. Specifically, a valuable study would be to systematically explore the realization of mission requirements (e.g., specified in natural textual language) in different behavior-modeling languages and measuring expressiveness, but also non-functional aspects such as succinctness and intuitiveness of the resulting model.

Also, we would like to provide good design guidelines for behavior-modeling languages that promote MDE practices but are compatible with the current state-of-practice. Finally, during our analysis we noticed that most projects lack mission specifications. We want to use our dataset to develop automatic tools to generate mission specifications in natural language from existing behavior-tree models. By achieving that we help in creating automated re-engineering tools for legacy projects that could be reused by other practitioners instead of starting from scratch.

Another valuable piece of future work would be to gather empirical data on the use of behavior-modeling languages in robotics projects in general, without the limitation to behavior-tree and state-machine DSLs. While these are, as we argue, the most prominent kinds of modeling languages used, we believe that other languages are also used. However, identifying other languages as used in open-source projects is not trivial and would constitute one or even multiple future studies.

ACKNOWLEDGMENTS

We thank Alexey Gorskiy for implementing the GitHub mining and model analysis scripts, including providing visualizations of SMACH and FlexBE scripts, usage statistics of the libraries, as well as extracting and calculating model metrics. Our work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

- [1] S. García, D. Strüber, D. Brugali, T. Berger, and P. Pelliccione, "Robotics software engineering: A perspective from the service robotics domain," in *FSE*, 2020.
- [2] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification patterns for robotic missions," *IEEE Trans. Software Eng.*, vol. 47, no. 10, pp. 2208–2224, 2021. [Online]. Available: <https://doi.org/10.1109/TSE.2019.2945329>
- [3] S. Dragule, S. Garcia, T. Berger, and P. Pelliccione, "Languages for specifying missions of robotic applications," in *Software Engineering for Robotics*, A. Cavalcanti, B. D. ad Rob Hierons, J. Timmis, and J. Woodcock, Eds. Springer, 2021.
- [4] F. Michaud and M. Nicolescu, "Behavior-based systems," in *Springer handbook of robotics*. Springer, 2016, pp. 307–328.
- [5] D. Kortenkamp, R. Simmons, and D. Brugali, "Robotic systems architectures and programming," in *Springer handbook of robotics*. Springer, 2016, pp. 283–306.
- [6] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part ii)," *IEEE Robotics & Automation Magazine*, vol. 17, no. 1, pp. 100–112, 2010.
- [7] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018.
- [8] J. Chen and D. Shi, "Development and composition of robot architecture in dynamic environment," in *RCAE*, 2018.
- [9] F. W. Heckel, G. M. Youngblood, and N. S. Ketkar, "Representational complexity of reactive agents," in *CIG*, 2010.
- [10] D. Isla, "Handling complexity in the Halo 2 AI," *GDC 2005 Proceedings*, 2005. [Online]. Available: https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php?page=2
- [11] K. Mcquillan, "A survey of behaviour trees and their applications for game AI," 2015, course CP5330 final report, James Cook University.
- [12] O. Biggar, M. Zamani, and I. Shames, "An expressiveness hierarchy of behavior trees and related architectures," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5397–5404, 2021.
- [13] M. Iovino, J. Förster, P. Falco, J. J. Chung, R. Siegwart, and C. Smith, "On the programming effort required to generate behavior trees and finite state machines for robotic applications," *arXiv preprint arXiv:2209.07392*, 2022.
- [14] J. A. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard *et al.*, "An integrated system for autonomous robotics manipulation," in *IROS*, 2012.
- [15] M. Colledanchise, A. Marzinotto, D. V. Dimarogonas, and P. Ögren, "The advantages of using behavior trees in multi robot systems," in *47th International Symposium on Robotics (ISR)*, 2016.
- [16] 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, 2016.
- [17] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *IEEE Transactions on Games*, vol. 11, no. 2, pp. 183–189, 2018.
- [18] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *ICRA*, 2014.
- [19] P. Ögren, "Increasing modularity of uav control systems using computer game behavior trees," in *AIAA Guidance, Navigation, and Control Conference (GNC)*, 2012.
- [20] F. Rovida, B. Grossmann, and V. Krüger, "Extended behavior trees for quick definition of flexible robotic tasks," in *IROS*, 2017.
- [21] S. García, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "High-level mission specification for multiple robots," in *SLE*, 2019.
- [22] A. Klöckner, "Interfacing behavior trees with the world using description logic," in *AIAA Guidance, Navigation, and Control Conference (GNC)*, 2013.
- [23] A. Wasowski and T. Berger, *Domain-specific Languages: Effective Modeling, Automation, and Reuse*. Springer, 2023, <http://dsl.design>.
- [24] E. de Araújo Silva, E. Valentin, J. R. H. Carvalho, and R. da Silva Barreto, "A survey of model driven engineering in robotics," *Journal of Computer Languages*, vol. 62, p. 101021, 2021.
- [25] G. L. Casalaro, G. Cattivera, F. Cicciozzi, I. Malavolta, A. Wortmann, and P. Pelliccione, "Model-driven engineering for mobile robotic systems: a systematic mapping study," *Software and Systems Modeling*, pp. 1–31, 2021.
- [26] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *SIMPAR*, 2014.
- [27] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "ROS: an open-source robot operating system," in *ICRA workshop on open source software*, 2009.
- [28] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wasowski, "Behavior trees in action: a study of robotics applications," in *SLE*, 2020.
- [29] N. Bencomo, R. B. France, B. H. C. Cheng, and U. Aßmann, Eds., *Models@run.time—Foundations, Applications, and Roadmaps*, vol. 8378. Springer, 2014.
- [30] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.
- [31] "Online appendix," <https://bitbucket.org/easelab/behaviortrees>, 2020. [Online]. Available: <https://doi.org/10.5281/zenodo.7515222>
- [32] M. Colledanchise, "Behavior trees in robotics," Ph.D. dissertation, KTH Royal Institute of Technology, 2017.
- [33] S. Dragule, T. Berger, C. Menghi, and P. Pelliccione, "A survey on the design space of end-user-oriented languages for specifying

- robotic missions," *Software and Systems Modeling*, vol. 20, pp. 1123–1158, 2021.
- [34] M. Colledanchise and P. Ögren, "How behavior trees generalize the teleo-reactive paradigm and and-or-trees," in *IROS*, 2016.
- [35] D. C. Conner and J. Willis, "Flexible navigation: Finite state machine-based integrated navigation and control for ros enabled robots," in *SoutheastCon*, 2017.
- [36] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The Marathon 2: A navigation system," in *IROS*, 2020.
- [37] J. M. Zutell, D. C. Conner, and P. Schillinger, "Ros 2-based flexible behavior engine for flexible navigation," in *SoutheastCon*, 2022.
- [38] S. Garcia, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "Promise: High-level mission specification for multiple robots," in *ICSE Demonstrations*, 2020.
- [39] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *ICCPs*, 2018.
- [40] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.
- [41] "ScenarioRunner for CARLA documentation," <https://carla-scenariorunner.readthedocs.io>, 2022.
- [42] M. Iovino, E. Scukins, J. Styrod, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and AI," *Robotics and Autonomous Systems*, vol. 154, p. 104096, 2022.
- [43] M. Colledanchise and L. Natale, "Analysis and exploitation of synchronized parallel executions in behavior trees," *arXiv preprint arXiv:1908.01539*, 2019.
- [44] I. Millington and J. Funge, *Artificial intelligence for games*. CRC Press, 2009.
- [45] M. L. Crane and J. Dingel, "UML vs. classical vs. rhapsody statecharts: not all models are created equal," *Software & Systems Modeling*, vol. 6, no. 4, pp. 415–435, 2007.
- [46] M. von der Beeck, "A comparison of statecharts variants," in *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1994.
- [47] Object Management Group, "OMG unified modeling language 2.5.1," 2017. [Online]. Available: <https://www.omg.org/spec/UML/>
- [48] B. P. Douglass, "Chapter 5 - design patterns for state machines," in *Design Patterns for Embedded Systems in C*, B. P. Douglass, Ed. Newnes, 2011, pp. 257–356.
- [49] M. Samek, "A crash course in UML state machines," *Quantum Leaps, LLC*, 2009.
- [50] H. Kubátová, K. Richta, and T. Richta, "Petri nets versus UML state machines," in *Proc. SDOT*, 2013, pp. 53–59.
- [51] B. Hajji, A. Mellit, and L. Bouselham, *Finite State Machines*. Singapore: Springer Singapore, 2022, pp. 175–205.
- [52] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.
- [53] G. Lüttgen, M. Von der Beeck, and R. Cleaveland, "A compositional approach to statecharts semantics," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 6, pp. 120–129, 2000.
- [54] D. Stonier, N. Usmani, and M. Staniaszek, "Py Trees library documentation," <https://py-trees.readthedocs.io/en/devel/>, 2020.
- [55] D. Faconti and M. Colledanchise, "BehaviorTree.CPP library documentation," <https://www.behaviortree.dev>, 2018.
- [56] J. Bohren and S. Cousins, "The SMACH high-level executive [ros news]," *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.
- [57] P. Schillinger, S. Kohlbrecher, and O. von Stryk, "Human-robot collaborative high-level control with an application to rescue robotics," in *IEEE International Conference on Robotics and Automation (ICRA)*, May 2016.
- [58] P. Schillinger, "An approach for runtime-modifiable behavior control of humanoid rescue robots," *Technische Universität Darmstadt*, 2015.
- [59] S. Kohlbrecher, A. Stumpf, A. Romay, P. Schillinger, O. Von Stryk, and D. C. Conner, "A comprehensive software framework for complex locomotion and manipulation tasks applicable to different types of humanoid robots," *Frontiers in Robotics and AI*, vol. 3, p. 31, 2016.
- [60] J. Bohren and S. Cousins, "SMACH library documentation," <http://wiki.ros.org/smach/Documentation>, 2010.
- [61] P. Schillinger, "FlexBE library documentation," <http://philserver.bplaced.net/fbe/documentation.php>, 2016.
- [62] D. Stonier, N. Usmani, and M. Staniaszek, "Py Trees ROS library documentation," <https://py-trees-ros.readthedocs.io/en/devel/>, 2020.
- [63] D. Faconti and M. Colledanchise, "BehaviorTree.CPP library tutorials," https://www.behaviortree.dev/tutorial_01_first_tree/, 2018.
- [64] D. Stonier, N. Usmani, and M. Staniaszek, "Py Trees ROS library tutorials," <https://py-trees-ros-tutorials.readthedocs.io/en/devel/tutorials.html>, 2020.
- [65] P. Schillinger, "FlexBE library tutorials," <http://wiki.ros.org/flexbe/Tutorials>, 2021.
- [66] J. Bohren and S. Cousins, "SMACH library tutorials," <http://wiki.ros.org/smach/Tutorials>, 2021.
- [67] D. Faconti, "MOOD2Be: Models and tools to design robotic behaviors," European Union's Horizon 2020 Research and Innovation Programme, 2019. [Online]. Available: https://github.com/BehaviorTree/BehaviorTree.CPP/blob/master/MOOD2Be_final_report.pdf
- [68] M. Colledanchise and L. Natale, "On the implementation of behavior trees in robotics," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5929–5936, 2021.
- [69] A. Romay, S. Kohlbrecher, A. Stumpf, O. von Stryk, S. Maniatopoulos, H. Kress-Gazit, P. Schillinger, and D. C. Conner, "Collaborative autonomy between high-level behaviors and human operators for remote manipulation tasks using different humanoid robots," *Journal of Field Robotics*, vol. 34, no. 2, pp. 333–358, 2017.
- [70] P. Schillinger, "An approach for runtime-modifiable behavior control of humanoid rescue robots," Master's thesis, TU Darmstadt, 2015, https://www.sim.informatik.tu-darmstadt.de/publ/da/2015_Schillinger_MA.pdf.
- [71] M. AlMarzouq, A. AlZaidan, and J. AlDallal, "Mining GitHub for research and education: challenges and opportunities," *International Journal of Web Information Systems*, 2020.
- [72] I. Malavolta, G. A. Lewis, B. Schmerl, P. Lago, and D. Garlan, "Mining guidelines for architecting robotics software," *Journal of Systems and Software*, vol. 178, p. 110969, 2021.
- [73] G. Robles, T. Ho-Quang, R. Hebig, M. R. Chaudron, and M. A. Fernandez, "An extensive dataset of UML models in GitHub," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 519–522.
- [74] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.
- [75] T. Berger and J. Guo, "Towards system analysis with variability model metrics," in *VaMoS*, 2014.
- [76] J. A. Cruz-Lemus, M. Genero, and M. Piattini, "Using controlled experiments for validating UML statechart diagrams measures," in *Software Process and Product Measurement*. Springer, 2007, pp. 129–138.
- [77] —, "Metrics for UML statechart diagrams," in *Metrics for Software Conceptual Models*, 2005, pp. 237–272.
- [78] R. van Tonder and C. Le Goues, "Lightweight multi-language syntax transformation with parser parser combinators," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 363–378.
- [79] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Pelliccione, and T. Berger, "Software variability in service robotics," *Empirical Software Engineering*, vol. 28, no. 2, pp. 1–67, 2023.
- [80] S. Garcia, D. Strueber, D. Brugali, A. D. Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability modeling of service robots: Experiences and challenges," in *VaMoS*, 2019.
- [81] D. Brugali and E. Prassler, "Software engineering for robotics [from the guest editors]," *IEEE Robotics & Automation Magazine*, vol. 16, no. 1, pp. 9–15, 2009.
- [82] I. A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I.-H. Shu *et al.*, "CLARAty: Challenges and steps toward reusable robotic software," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 5, 2006.
- [83] M. Colledanchise, "BT++ library documentation," <https://github.com/miccol/ROS-Behavior-Tree/blob/master/BTUserManual.pdf>, 2017.
- [84] F. Rovida, "SkiROS2 library documentation," <https://github.com/RVMI/skiros2/wiki>, 2020.
- [85] RoboSoft AI, "SMACC library documentation," <https://smacc.dev/>, 2018.

- [86] M. Steinbrink, P. Koch, S. May, B. Jung, and M. Schmidpeter, "State machine for arbitrary robots for exploration and inspection tasks," in *Proceedings of the 2020 4th International Conference on Vision, Image and Signal Processing*, 2020, pp. 1–6.
- [87] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–10.
- [88] F. Rovida, M. Crosby, D. Holz, A. S. Polydoros, B. Großmann, R. Petrick, and V. Krüger, "SkiROS — a skill-based robot control platform on top of ROS," in *Robot operating system (ROS)*. Springer, 2017, pp. 121–160.
- [89] F. Rovida and V. Krüger, "Design and development of a software architecture for autonomous mobile manipulators in industrial environments," in *ICIT*, 2015.
- [90] P. Laker, "Blackboard design pattern," <https://social.technet.microsoft.com/wiki/contents/articles/13215-blackboard-design-pattern.aspx>, 2012.
- [91] J.-P. Tolvanen and S. Kelly, "How domain-specific modeling languages address variability in product line development: Investigation of 23 cases," in *23rd International Systems and Software Product Line Conference*, ser. SPLC, 2019.
- [92] A. Alami, Y. Dittrich, and A. Wasowski, "Influencers of quality assurance in an open source community," in *11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2018.
- [93] E. F. Moore *et al.*, "Gedanken-experiments on sequential machines," *Automata studies*, vol. 34, pp. 129–153, 1956.
- [94] J. Bosch, "Design patterns as language constructs," *Journal of Object Oriented Programming*, 1997.
- [95] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2013.
- [96] J. A. Cruz-Lemus, A. Maes, M. Genero, G. Poels, and M. Piattini, "The impact of structural complexity on the understandability of UML statechart diagrams," *Information Sciences*, vol. 180, no. 11, pp. 2209–2220, 2010.
- [97] M. Genero, D. Miranda, and M. Piattini, "Defining and validating metrics for UML statechart diagrams," in *QAOOSE*, 2002.
- [98] J. A. Cruz-Lemus, M. Genero, M. Piattini, and A. Toval, "Investigating the nesting level of composite states in UML statechart diagrams," *Proc. QAOOSE*, vol. 5, pp. 97–108, 2005.
- [99] —, "An empirical study of the nesting level of composite states within UML statechart diagrams," in *International Conference on Conceptual Modeling*. Springer, 2005, pp. 12–22.
- [100] D. Brugali and P. Scandurra, "Component-based robotic engineering (part i)[tutorial]," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, 2009.
- [101] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *CSMR*, 2013.
- [102] L. Gherardi and D. Brugali, "Modeling and reusing robotic software architectures: The hyperflex toolchain," in *ICRA*, 2014.
- [103] C. Schlegel, A. Lotz, M. Lutz, D. Stampfer, J. F. Inglés-Romero, and C. Vicente-Chicote, "Model-driven software systems engineering in robotics: covering the complete life-cycle of a robot," *Information Technology*, vol. 57, no. 2, pp. 85–98, 2015.
- [104] D. L. Wigand, A. Nordmann, M. Goerlich, and S. Wrede, "Modularization of domain-specific languages for extensible component-based robotic systems," in *IRC*, 2017.
- [105] SPARC Robotics, "Robotics 2020 multi-annual roadmap for robotics in Europe," *SPARC Robotics, EU-Robotics AISBL, The Hague, The Netherlands*, 2017.



Razan Ghzouli is currently working towards a PhD degree at Chalmers University of Technology, Gothenburg, Sweden, where she is part of the software engineering division at the Department of Computer Science and Engineering. She received her master degree in applied data science from the university of Gothenburg, Sweden and her bachelor degree in computer and automation engineering from Damascus university, Syria. Her PhD focuses on facilitating the migration to model-based design and systems to

enable reusable and maintainable robotic missions.



Thorsten Berger is a Professor in Computer Science at Ruhr University Bochum in Germany. After receiving the PhD degree from the University of Leipzig in Germany in 2013, he was a Postdoctoral Fellow at the University of Waterloo in Canada and the IT University of Copenhagen in Denmark, and then an Associate Professor jointly at Chalmers University of Technology and the University of Gothenburg in Sweden. He received competitive grants from the Swedish Research Council, the Wallenberg Autonomous Systems Program, Vinnova Sweden (EU ITEA), and the European Union. He is a fellow of the Wallenberg Academy—one of the highest recognitions for researchers in Sweden. He received two *best-paper* and two *most-influential-paper* awards. His service was recognized with distinguished reviewer awards at the tier-one conferences ASE 2018 and ICSE 2020, and at SPLC 2022. His research focuses on model-driven software engineering, program analysis, and empirical software engineering.



Einar Broch Johnsen is a Professor at the Department of Informatics of the University of Oslo in Norway. He is the strategy director of Sirius, a center for research-driven innovation with long-term funding from the Research Council of Norway. He has been prominently involved in many national and European research projects; in particular, he was the coordinator of the EU FP7 project Envisage (2013-2016) on formal methods for cloud computing and the scientific coordinator of the EU H2020 project HyVar (2015-2018) on hybrid variability systems. His research focuses on formal methods, programming models and methodology, and model-based analysis in domains such as distributed and concurrent systems, cloud computing, digital twins and robotics.



Andrzej Wasowski is Professor of Software Engineering at the IT University of Copenhagen. He has also worked at Aalborg University in Denmark, and as visiting professor at INRIA Rennes and University of Waterloo, Ontario. His interests are in software quality, reliability, and safety in high-stake high-value software projects. This includes semantic foundations and tool support for model-driven development, program analysis tools, testing tools and methods, as well as processes for improving and maintain quality in software projects. Many of his projects involve commercial or open-source partners, primarily in the domain of robotics and safety-critical embedded systems. Recently he coordinates the Marie-Curie training network on Reliable AI for Marine Robotics (REMARO). Wasowski holds a PhD degree from the IT University of Copenhagen, Denmark (2005) and a MSC Eng degree from the Warsaw University of Technology, Poland (2000).



Swaib Dragule is a PhD Fellow in Computer Science and Software Engineering at Chalmers University of Technology and Makerere University. He holds MSc. and BSc. in computer science. He is an academic staff of Makerere university, College of Computing and Information Sciences. His research interests are in programming languages, domain-specific languages, and robotics.