COMPLEX SYSTEM MODELING AND SIMULATION
ISSN 2096-9929 06/06 pp 357–380
Volume 3, Number 4, December 2023
DOI: 10.23919/CSMS.2023.0017

Towards Efficient Robotic Software Development by Reusing Behavior Tree Structures for Task Planning Paradigms

Shuo Yang and Qi Zhang*

Abstract: Nowadays, autonomous robots are expected to accomplish more complex tasks and operate in an open-world environment with uncertainties. Developing software for such robots involves the design of task planning paradigms and the implementation of robotic software architectures, making software development rather tricky and time-consuming. In recent decades, component-based software development approaches have been increasingly adopted in robotics to improve software development efficiency by reusing data and controlling flows between components. However, few works have tackled the more critical issue of reusing complex high-level task planning paradigms and robotic software architectures. To make up for the limitation, this paper first identifies the mainstream task planning paradigms and proposes a set of novel patterns for interaction pipelines between the robotic functions of sensing, planning, and acting. Then this paper presents a novel Behavior Tree (BT) based development framework Structural-BT, which provides a set of reusable BT structures that implement abstract interaction pipelines while maintaining interfaces for task-specific customization. The Structural-BT framework supports the modular design of structure functionalities and allows easy extensibility of the inner planning flows between BT components. With the Structural-BT framework, software engineers can develop robotic software by flexibly composing BT structures to formulate the skeleton software architecture and implement task-specific algorithms when necessary. In the experiment, this paper develops robotic software for diverse task scenarios and selects the baseline approaches of Robot Operating System (ROS) and classical BT development frameworks for comparison. By quantitatively measuring the reuse frequencies and ratios of BT structures, the Structural-BT framework has been shown to be more efficient than the baseline approaches for robotic software development.

Key words: robotic software modeling and development; software architecture; task planning paradigm; behavior tree modeling

1 Introduction

Autonomous robots have been playing an increasingly important role in the open-world human society, accomplishing various challenging tasks such as field exploration, remote transportation, and domestic service. Open-world environments commonly feature state dynamics and perception uncertainties, making it much more difficult for autonomous robots to achieve the tasks successfully. The increasing complexity of robot task requirements and environmental uncertainties makes developing the underlying autonomous robot software cumbersome, which has drawn much attention from both robotics and software engineering communities[1].

Autonomous robot software, commonly conceived as

• Shuo Yang and Qi Zhang are with the College of Systems Engineering, National University of Defense Technology, Changsha 410072, China. E-mail: zhangqiy123@nudt.edu.cn.
∗ To whom correspondence should be addressed.
※ This article was recommended by Associate Editor Wenyin Gong.
Manuscript received: 2023-04-21; revised: 2023-06-25; accepted: 2023-07-22

© The author(s) 2023. The articles published in this open access journal are distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).

the medium to embody intelligence and deployed upon the robotic hardware[2, 3], plays a critical role in making high-level plans for achieving tasks and driving hardware devices for concrete plan execution. The software generally consists of an abstract layer that regulates a high-level task planning paradigm for task achievement, and a concrete layer that architects interaction pipelines between concrete software components in the software architecture. The abstract layer establishes the high-level task planning paradigm that specifies how the computational functions of planning, sensing, and acting interact to make up task plans. Different robotic tasks require diverse different task planning paradigms under specific environmental assumptions. Robot software development must decide the top-level task planning paradigm based on specific tasks and environment characteristics. The concrete layer maintains a styled software architecture that specifies how software is divided into different functional software components and how the data, planning, and controlling flows run through these components. The task planning paradigm and robot software architecture are closely interconnected and mutually influenced. On one side, the abstract task planning paradigm generally decides the design choice of underlying software architecture. Conversely, the robotic software architecture is commonly regarded as the concrete implementation solutions regarding the components' organizational structures that support the high-level task planning paradigms. Software developers must make an integral architecture decision and make trade-offs when necessary[4]. In this sense, the development of autonomous robot software needs to integrally consider the designs of the above two layers for both the algorithmic achievements of task goal and architectural structure of software components[5].

The autonomous robot software, commonly conceived as a complex cyber-physical system, urgently requires effective modeling and development approaches for efficient development[6]. In the past few decades, developing autonomous robotic software has long been recognized as a challenging issue for both robotic researchers and software engineers[3, 7, 8], especially for software in open-world environments. More specifically, there are three main challenges to current robotic software development practices. Firstly, developing task planning paradigms highly depends on developers' expertise and experiences due to the need for clearly specified design patterns. Different robotic tasks may feature the same tasks and environments, whose software development may require similar patterns of task planning paradigms. Software engineers' primary and challenging step is identifying and recognizing specific patterns for task planning paradigms[9, 10]. Secondly, current robotic software architectures' reuse granularity must be improved for high-level decision-making schemes. Existing robotic software architectures mainly provide basic reusable interfaces for data communications without considering complex controlling and planning flows between software components, resulting in low levels of software reuse in robotics. Thirdly, the robotic community still lacks an integral solution for robotic software development. Robotic software development needs to holistically consider the design choices of high-level task planning paradigms and concrete software architecture. The two layers of autonomous robot software are closely interconnected and influenced, which requires software engineers to take an integrated development solution to maintain layer consistency.

The challenges above have jointly lowered the efficiency of robotic software development, making such systems rather tricky and time-consuming to construct. To enable efficient robotic software development, this paper presents a novel component-based software development framework Structural-BT that increases the reuse granularity from component level to structure level. Generally, the main contributions of the Structural-BT framework are three-fold.

- **Abstract paradigm patterns for design reusability**. This framework formalizes the mainstream task planning paradigms and develops a set of reusable patterns regarding the interaction pipelines between sensing, planning, and acting functions. It is perhaps the first time these highly abstract paradigms are concretely formalized and decomposed. It would be much easier for software engineers to design high-level decision-making schemes by flexibly synthesizing the paradigm patterns. Structural-BT has been the first robotic component framework that enables design reusability of the high-level task planning paradigms, which greatly improves the reuse granularity from component to structure level.
- **Concrete BT structures for implementation reusability**. The framework provides a set of reusable

BT structures based on behavior tree components for robotic software architecture implementation, with each BT structure embedding the concrete algorithmic implementation of interaction schemes. Software engineers can efficiently implement the robotic software architectures by flexibly composing the concrete reusable BT structures and making customized task-specific modifications.

- **Validated efficiency of robot software development**. This paper has preliminarily validated the feasibility of paradigm-level reuse in robotic software development. The experiment results have proved the improved efficiency of robotic software development with Structural BT by comparing the reusability performance with Robot Operating System (ROS) and classical BT baseline approaches, two of the most popular robotic component based development frameworks.

2 Related Work

In recent years, robotic experts and software engineering researchers have made joint efforts to minimize robotic software development complexity and improve efficiency by introducing a set of practical software engineering techniques into robotics, including component-based[11, 12], model-driven[13, 14], and service-oriented[15] approaches. Notably, component-based software engineering has become one of the most popular and widely acknowledged approaches in robotics, which shifts the emphasis of robotic software development from traditional ad-hoc analysis and programming to composing existing reusable components[2]. In this section, we review existing component-based works and discuss the limitations of software reuse in robotic software development.

In the past few decades, the Robot Operating System (ROS) framework has been recognized as one of the most popular component-based software development frameworks in robotics[11, 16, 17]. It creatively solves heterogeneity by reusing software components and robotic algorithms across diverse robotic hardware platforms, establishing the defacto software development standard in robotic communities. The ROS framework proposes the ROS node component, which wraps the remote communication schemes of "topic" and "service" as node infrastructures, facilitating easy reuse of communication-related code snippets. With standardized communication interfaces and independent node functionalities, robotic software can be quickly developed by composing a set of functional ROS nodes without much effort in programming the data communication codes. For example, Xin et al.[18] utilized the ROS node components and its distributed architecture for implementing the distributed model predictive control of multi-robot systems.

Recently, the Behavior Tree (BT)[19–21] framework has been newly introduced into robotics as another promising component-based development approach. The BT framework has provided a set of functional BT node components, including internal control flow nodes encapsulating the common control logics of sequential, parallel, and fallback, and external execution nodes that abstract the durative robotic acting action and instantaneous condition node. Technically, BT robotic software is built by composing diverse types of BT nodes. The BT framework enables good modular design on all scales ranging from the topmost subtree structures to all tree leaf nodes. The BT framework has gained popularity in robotics due to its enhanced reusability of common controlling logic in the control flow nodes, as most robotic software needs to handle the implementation of the common controlling logic. The BT framework has increased the level of reusability beyond that of the ROS framework by encapsulating and reusing the robotic controlling schemes, not just inter-node data communication channels. Many works are introducing the BT framework into robotics. In Ref. [21], Kuckling et al. explored the possibility of adopting behavior trees as an architecture for the control software of robot swarms. They introduced Maple's automatic design method to combine preexisting modules into behavior trees. In Ref. [22], Woolley and Peterson presented a unified behavior tree software framework representing five famous behavior-based control structures. The unified behavior tree framework (1) eases the complexity of development and testing; (2) promotes code reuse; (3) supports designs that scale quickly into large hierarchies of focused base behaviors, and (4) allows system developers with the freedom to use the architectures that function the best. In Ref. [23], Yang et al. utilized BT components to implement an adjoint observation scheme by proposing parallel and fallback BT tree structures. The authors extended BT control nodes with an online planning component and mutual data store mechanism, enabling continuous planning

and efficient data communication between robotic sensing and actuating processes.

Besides the two well-known component frameworks above, robotic software engineers have proposed different component models for robotic software development. For example, in Ref. [24], Cassou et al. proposed a new notion of interaction contracts that specify the allowed interactions between components for developing Sense-Compute-Control (SCC) application software. SCC robotic software systems can be designed and implemented by an architectural pattern that includes four components (sensors, context operators, control operators, and actuators) and the reuse of interaction contracts. Experiments show that interaction contracts significantly improve program size, execution coverage, and code quality. In Ref. [25], Bruyninckx et al. developed methods and tools in the BRICS project to design, configure, and compose stable robotic software architectures. Each functional sub-system is designed as a software product line whose architecture explicitly models software variation points and variants.

As can be analyzed, both the ROS and classical BT frameworks have shown limitations in meeting the three challenges above, which leaves the research gap in efficient robotic software development. Firstly, neither ROS nor the classical BT framework offers the easy-to-follow design patterns of robotic software architecture, with only essential node models and communication mechanisms provided. Secondly, both frameworks have limited the reusability level to node-level or control unit-level, without basic reusable infrastructures for higher-level robotic decision-making paradigms. Thirdly, the above works have rarely proposed the reusability concepts integrally from the full cycle of software design to software implementation, with most works concerning on partial phase of robotic software development.

3 Structural-BT Development Framework

The Structural-BT software development framework aims to improve robotic software development efficiency by presenting the conceptual reuse design of task planning paradigms and providing a set of concrete reusable BT structures. In this section, we first illustrate our insights on the hierarchical abstraction of robotic software development procedures, which explicitly specifies the abstract layer of decision-making logic and a concrete layer of architecture design for the first time. Then we illustrate the design motivation of structural-reuse setting in the Structural-BT framework. The following two sections present the technical details regarding the task planning paradigms abstraction and reusable BT structures development.

3.1 Insights on robotic software development

As previously discussed, the robotic software can be explicitly abstracted and decomposed into two interconnected layers to separate concerns. In this paper, we propose that robotic software development should be handled in a multi-phase workflow that separates the abstract paradigm design from the concrete architecture implementation, which helps to reduce the overall complexity of software development and offers a systematic engineering solution. The current robotic software engineering practices treat software development as a miscellaneous and ad-hoc coding process for achieving the task requirements without recognizing the diverse layers of software concerns. Such an intuitive development style may be inefficient for complex robotic software with deliberative task requirements and environmental uncertainties as the design complexity and difficulty of task planning paradigms and software architectures increase rapidly. Therefore, the Structural-BT framework dedicates to tackling the following two issues:

- **Ad-hoc design of task planning paradigm.** The interaction design for each computational function in a task planning paradigm is essentially ad-hoc and cumbersome. The current software engineering approaches require the developer to manually design the interaction pipelines (sensing-planning, planning-acting, and sensing-acting) and construct the software prototype mostly based on their expertise. There are no fixed or reusable interaction patterns provided for novice developers. It may be difficult for novice developers to decide on a suitable task planning paradigm quickly, so they usually fail to reuse existing good designs from other software products.

- **Inadequate reusability level of robotic software architecture.** Reusable designs in existing robotic software architectures (such as ROS and BT) are generally limited to the data and controlling logic level without considering the more complex task planning logic, making developing task planning loops in robotic software difficult and cumbersome. Software developers can reuse the ROS topic/service remote communication schemes in their ROS-based software

architecture without manually developing the component communication channels. However, software developers program the controlling and planning flow in these node components without reusing existing solutions, which generally requires much development effort.

3.2 Motivation of Structural-BT framework design

Motivated by the above goals, the design of the Structural-BT framework dedicates to improving the software reuse granularity from the current component level to the structure level, which expects to reduce the design complexity and improve software development efficiency. Figure 1 presents the design of the Structural-BT framework for structure-level software reuse. The core reuse design of the Structural-BT framework adopts a "decompose-realize-compose" process, with each step explained as follows:

- **Decompose**. This step decomposes the current mainstream robotic task planning paradigms into fundamental paradigm baselines. By abstracting and decomposing the critical functions of these paradigms, we can recognize some commonly used functions and their interaction patterns, which resolves the ad-hoc design issue of task planning paradigms. In the Structural-BT framework, we have acquired three general patterns of interaction pipelines (Sensing & Planning, Sensing & Acting, and Planning & Acting) between the critical sensing, planning, and acting functions, which could be further reused for developers to design customized task planning paradigms for any robotic tasks.
- **Realize**. After identifying the above abstract interaction patterns, we propose to realize them as concrete software artifacts so that the identified paradigm patterns can be practically reused. Our approach uses the behavior tree component framework to represent and realize them as a set of reusable behavior tree structures.
- **Compose**. the reusable behavior tree structures implemented with standardized composable interfaces also allow easy algorithmic customization in task-relevant behavior tree components. The developer could easily compose one or several reusable structures in this setting to produce a preliminary and prototypical robotic software architecture. The "Realize" and "Compose" steps have jointly improved the reusability level from the component-level to the structure-level, which resolves the second reusability issue of robotic software architecture.

4 Abstraction of Task Planning Paradigms

In this section, we identify existing mainstream task planning paradigms in robotics and explore how to reuse these abstract paradigms in robotic software development. We first identify the paradigms from existing literature and open-sourced robotic software projects. Then we establish the symbolic representation of each computational function and its interaction pipelines to synthesize fixed interaction patterns in these paradigms. Finally, we validate the feasibility and applicability of the proposed interaction patterns by illustrating the composition algorithms that formulate the existing task planning paradigms in robotic

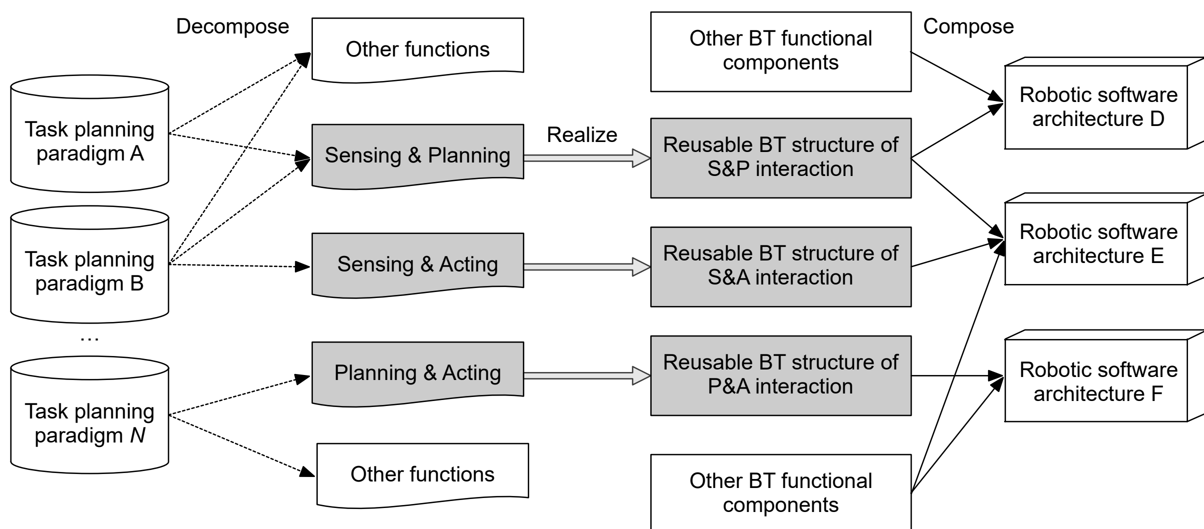


Fig. 1 Overview of structure-level reuse design in the Structural-BT framework.

software development.

4.1 Mainstream task planning paradigms

We obtained a set of influential research papers in robotics and software engineering disciplines and 25 popular robotic software projects from the open-source GitHub repositories to identify the mainstream task planning paradigms in robotics. To obtain as many resources as possible, we used the following search strings to collect the academic papers and projects: robot task planning and robot planning (control paradigm or control loop or task planning algorithm) in the Google Scholar search engine. The collected academic papers[9, 26–30] and software projects generally have highly significant effects and have gained much popularity within the robotic community, so they serve as reliable foundations for our analysis results. We have extracted five mainstream robotic task planning paradigms: hierarchical, reactive, hybrid, and online/offline probabilistic. Figures 2−6 have graphically illustrated the interaction pipelines between the sensing, planning, and acting functions for the five paradigms.

The hierarchical planning paradigm[28], called the sense-plan-act paradigm, maintains sequential interaction pipelines. The sensing function first senses the environment and sends sensor-based knowledge (domain model) to the planning function. Then the planning function outputs the task plan to the acting function for execution. In this paradigm, three computational functions decompose the task planning problem into vertical slices, making the information flow from the environment via the sensing function and back to the environment via the acting function. The hierarchical planning paradigm establishes the closed

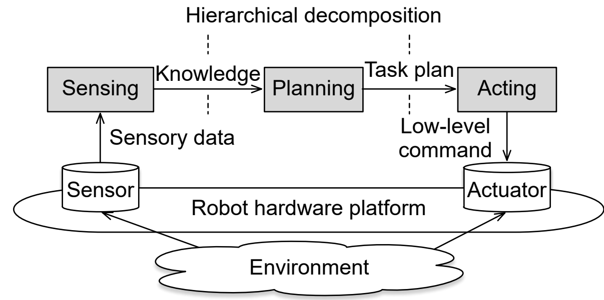


Fig. 2 Interaction pipeline of the hierarchical planning paradigm.

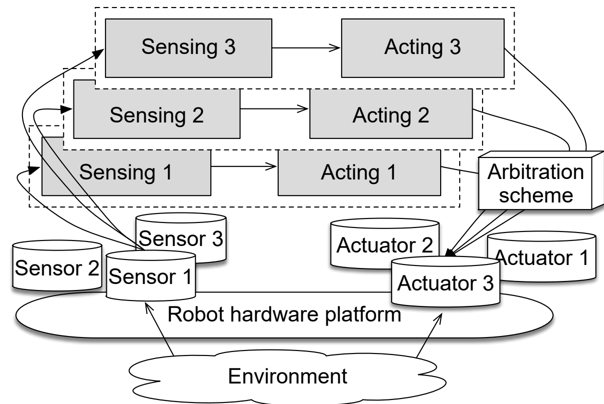


Fig. 3 Interaction pipeline of the reactive planning paradigm.

feedback loop and effectively solves robotic tasks in static and fully observable environments. Figure 2 graphically depicts the hierarchical decomposition of the three functions and the sequential information flow.

The reactive planning paradigm is one of the most effective planning solutions for robotic tasks in dynamic environments, which is popular in robotic software with real-time requirements[29]. This paradigm implements a direct input/output interaction pipeline

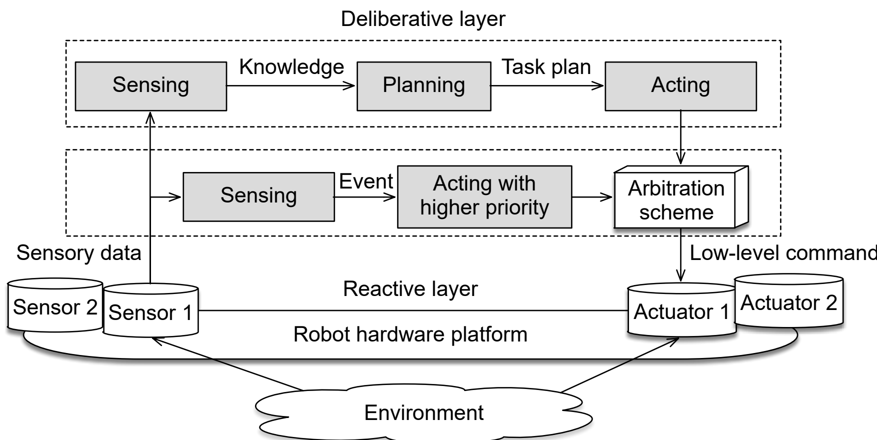


Fig. 4 Interaction pipeline of the hybrid planning paradigm.

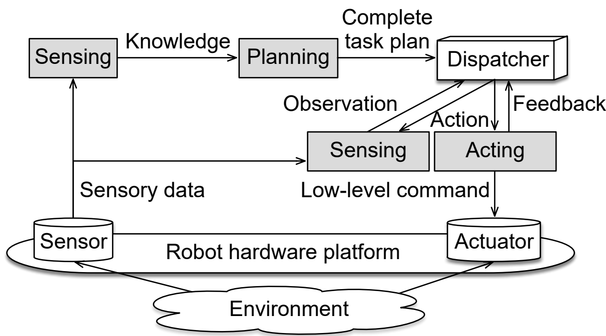


Fig. 5 Interaction pipeline of the offline probabilistic planning paradigm.

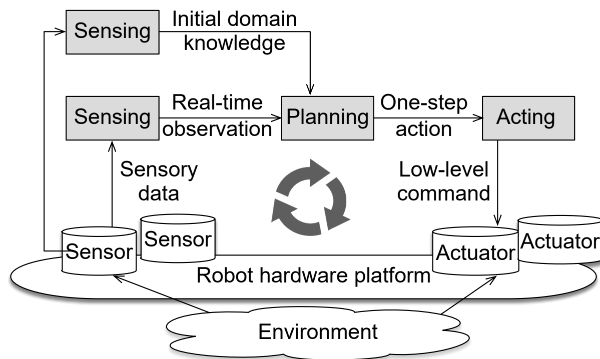


Fig. 6 Interaction pipeline of the online probabilistic planning paradigm.

from the sensing function to the acting function without intermediate computational functions for long-term deliberation. The sensing function receives sensory data from robot sensor devices and triggers the corresponding acting functions to handle the events based on pre-defined rules. Multiple sensing-acting interaction pipelines can be hierarchically organized, with each pipeline running at a specific priority level and being scheduled by user-specified arbitration schemes. This paradigm supports strongly reactive robot software that quickly responds to dynamic environment changes. Figure 3 presents a graphical representation of the multiple pipelines of direct interactions between sensing and acting functions in the reactive planning paradigm.

The hybrid planning paradigm[9, 28], also known as the layered planning paradigm, goes a step further to integrally consider both the goal deliberation and event reaction requirements of robotic tasks. This paradigm combines the hierarchical paradigm's deliberative planning capability and the reactive paradigm's strong reactivity. In this paradigm, the deliberative layer implements high-level deliberation by maintaining the typical sense-plan-act interaction pipeline that makes a deliberative task plan based on internal domain models. The reactive layer implements the reactive pipeline that directly couples sensory information from the sensing function to the acting function, allowing the robot to cope reactively with environment dynamics while making high-level task plans.

The probabilistic planning paradigms have been widely known for making robust plans in open-world environments with uncertain acting and perception[26, 27, 31]. The paradigms have utilized probabilistic planning models (such as Markov Decision Process (MDP) and Partially Observable Markov Decision Process (POMDP)) to model non-deterministic acting effects and partially observable perception results, enabling the robot task plan to be robust in any possible environment. Specifically, two probabilistic planning paradigms maintain offline and online planning pipelines for diverse environmental conditions.

The offline probabilistic planning paradigm suits robotic tasks under partially known environment states. The planning function receives initial domain knowledge from the sensing function and creates a complete task plan. The task plan has a contingent structure that contains sensing action operators for checking runtime environment conditions and acting action operators for changing environment states. The complete task plan gets executed and receives runtime observations and feedback from each dispatched action. The offline paradigm deals with the uncertainties of environment states by dynamically switching to the proper contingent branch based on the runtime observations from the sensing actions. Figure 5 depicts the offline paradigm's interaction pipeline and iterative plan dispatching process.

Unlike the offline paradigm, the online probabilistic planning paradigm makes a one-step action at each planning step. It iteratively performs the planning function until the task goal is finally achieved. The online paradigm repetitively performs the "sensing-planning-acting" interaction pipeline: the sensing function sends real-time observation results to the planning function, and the planning function computes the currently optimal action and sends it to the acting function for acting. This paradigm makes no complete plan at the initial stage. Instead, it incrementally outputs one-step actions at each planning step after receiving a new observation. Due to its real-time sensing and planning capabilities, the online paradigm

enables robust task achievement under possible environmental changes. Figure 6 presents the online paradigm's repetitive interaction pipeline of "sensing-planning-acting".

4.2 Common patterns of sensing, planning, and acting

In this section, we look deeper into the interaction pipelines of the above task planning paradigms and explore the common interaction patterns between these computational functions. We first formalize the critical concepts and functions of task planning paradigms and then propose a set of common interaction patterns for reuse in software prototype design.

4.2.1 Concept and definition

Definition 1 (Task planning paradigm) A common task planning paradigm contains a set of critical domain concepts and the computational functions of sensing, planning, and acting. We define a basic paradigm as a tuple $\Sigma = \langle S, A, Z, f_s, f_p, f_a \rangle$ where S is the set of all the environment states, A is the set of all possible actions, and Z is the set of all possible observations. f_s, f_p, and f_a represent the computation functions of sensing, planning, and acting, respectively.

Definition 2 (State and observation) A state $s_t \in S$ is a description of the properties and status of various objects in the robot's situated environment at time step t. An observation $z_t \in Z$ is the sensed knowledge based on the raw sensing readings from the sensors at the state s_t. This paper describes the state s_t and observation z_t at time step t as a set of ground atoms.

Definition 3 (Action) An action operator represents the behaviors of a robotic sensor or actuator device. In the view of robotic software, a sensor device (such as a camera or laser sensor) senses the environment and receives environmental information. The operation of a sensor device does not change the environmental state. However, an actuator device (such as an arm or move base) changes the environment states by its operation effects but receives no sensory information. Based on this characteristic, we abstract two types of action operators: sensing actions and actuating actions. Formally, a sensing action operator is a tuple $a_s = \langle \tau, \upsilon \rangle$, where τ is the target to be sensed. The sensing target τ may either be an environmental object that needs to be recognized or an environmental condition that needs to be observed. υ represents the received observation after action execution. An actuating action operator is a tuple $a_a = \langle \varpi, \epsilon \rangle$, where ϖ is the precondition that makes the action applicable to execute, and ϵ represents the effects of the actuating action.

Definition 4 (Planning task) A robotic task generally requires a robot to fulfill a specific goal from the initial state. We define a planning task Π as $\Pi = \langle I, G, K \rangle$, in which:

- $I = s_t$ is the environment state at initial time step t;
- $G = s_{t'}$ is the goal environment state at time step t';
- $K = \langle z_t, A \rangle$ is the initially sensed domain-specific knowledge z_t regarding the initial environment state s_t and the set of available actions A that the robot can operate.

A task plan Δ is a generous concept that refers to the output from the planning function, which can be defined as an ordered set of action operators $\Delta = \{a_0, a_1, \ldots, a_n\}$, where a_i may either be a sensing a_s or actuating a_a action operator. A task plan Δ for a planning task Π can be formalized as $I \xrightarrow{\Delta} G$, representing that the plan Δ is capable of transitioning the environment from the initial state I to the goal state G.

Definition 5 (Planning function) The planning function f_p is the process that solves a planning task $\Pi = \langle I, G, K \rangle$ by making up a valid task plan Δ. We formalize the planning function f_p as

$$f_p(\Pi) = \begin{cases} \Delta, & I \xrightarrow{\Delta} G; \\ \text{failure}, & \text{otherwise}. \end{cases}$$

Definition 6 (Acting function) The acting function f_a is dedicated to executing an actuating action a_a, which transitions the environment state with the action effects. Assume the robot needs to execute action a_a at t on the current environment state s_t, the acting function f_a goes as follows:

$$f_a(s_t, a_a) = s_{t+1} = s_t \cup a_a(\epsilon), s_t \models a_a(\varpi).$$

When current state s_t satisfies the action precondition $a_a(\varpi)$, the resulting environment state s_{t+1} is then formulated by merging the action effects $a_a(\epsilon)$ with state s_t.

Definition 7 (Sensing function) The sensing function f_s is the process of executing a sensing action a_s that aims to sense the target $a_s(\tau)$ based on the current environment state s_t and receives observation z_t after execution. Notably, the observation z_t may be either the complete or partial sensed knowledge upon the state s_t depending on the full or partial

observability of environment states. The function can be defined as

$$f_s(s_t, a_s) = a_s(\upsilon) = z_t \quad (1)$$

4.2.2 Interaction pipelines between functions

The above concepts and definitions have described the independent functionalities of sensing, planning, and acting functions. In this section, we examine the standard interaction pipelines between these functions, which helps decompose the complex task planning paradigms into a set of reusable interaction patterns.

Definition 8 (Interaction pipeline) An interaction pipeline is essentially the information flow between two computational functions. The information flows between different functions are generally diverse regarding the information types and temporal features. We first define a basic interaction pipeline as the tuple $\kappa = \langle f_1, f_2, D, T \rangle$, and these elements are explained as follows:

- $D = f_1 \rightarrow f_2 | f_1 \leftarrow f_2$, which indicates the information flow is initially generated from/to the computational function f_1 to/from f_2.
- $T = f_1 \ominus f_2 | f_1 \circledcirc f_2$, indicating that function f_1 interacts with f_2 at one-shot ($\ominus$) or periodical ($\circledcirc$) temporal styles.

Definition 9 (S-P interaction pipeline) The interaction pipeline κ_{sp} between the sensing f_s and planning f_p function is aimed at formulating sensed domain knowledge based on sensory information and making valid task plan based on the knowledge. In the pipeline, the information flows from the sensing function to the planning function, sending the received observation and knowledge as the planning input ($f_s \rightarrow f_p$). In the paradigms above, the interaction pipeline between sensing and planning functions is either one-shot ($\ominus$) in the hierarchical paradigm or periodically ($\circledcirc$) in the online probabilistic paradigm. In this case, the S-P interaction pipeline can be expressed as follows:

$$\kappa_{sp} = \langle f_s, f_p, f_s \rightarrow f_p, f_s \ominus f_p | f_s \circledcirc f_p \rangle.$$

Definition 10 (S-A interaction pipeline) The interaction pipeline κ_{sa} between the sensing f_s and acting f_a function aims to react to unexpected environmental changes by continually receiving the real-time sensory information and sending it to the acting function for quick response. In this pipeline, the sensory information flows directly from the sensing function to the acting function without passing through any deliberative planning functions ($f_s \rightarrow f_a$). The reactive planning paradigm consists of a collection of prioritized sensing-acting interaction pipelines, requiring these pipelines to run periodically to make the robot robust to any environmental changes. The S-A interaction pipeline can thus be described as follows:

$$\kappa_{sa} = \langle f_s, f_a, f_s \rightarrow f_a, f_s \circledcirc f_a \rangle.$$

Definition 11 (P-A/S interaction pipeline) The interaction pipeline κ_{pas} handles the issue of how to dispatch and execute a compete task plan. In the above paradigms, a complete task plan may have diverse structures, including a deterministic plan that contains a sequentially ordered set of actuating actions, and a non-deterministic plan that contains both sensing and actuating actions in a contingently branched structure. The above plans generally require a continuous and iterative loop of action dispatching and feedback until the task plan is executed completely. Notably, a deterministic plan generally requires the interaction pipeline κ_{pa} in which the actuating actions in the plan are dispatched to the acting function f_a and return the execution status. The non-deterministic plan requires the interaction pipeline κ_{pas} in which both actuating and sensing actions of the plan are dispatched to the acting f_a and sensing f_s functions, respectively. The sensing action returns the observation, and the task plan switches to the corresponding actuating action branch accordingly. In these cases, both interaction pipelines κ_{pa} and κ_{pas} need to work periodically. The task plan information always flows from the planning function's output to the input of the acting/sensing function. Therefore, the P-A/S interaction pipelines can be expressed as follows:

$$\kappa_{pa} = \langle f_p, f_a, f_p \rightarrow f_a, f_p \circledcirc f_a \rangle,$$
$$\kappa_{pas} = \langle f_p, f_a/f_s, f_p \rightarrow f_a/f_s, f_p \circledcirc f_a/f_s \rangle.$$

4.3 Composition of interaction pipelines

The sections above have formalized the computational functions and abstracted a set of common interaction pipelines between these functions. In this section, we compose the proposed interaction pipelines to formulate the above mainstream task planning paradigms to check the reasonability and reusability of these common interaction pipelines.

4.3.1 Hierarchical planning paradigm

As illustrated in Fig. 2, the hierarchical planning paradigm features the sequential execution of sensing, planning, and acting. This paradigm assumes a static environment and performs one-shot planning. The planning function makes a complete task plan

consisting of a set of ordered action operators, which must be continuously dispatched until the last action operator gets executed. Based on the above features, the algorithmic description of the hierarchical planning paradigm is shown in Algorithm 1.

4.3.2 Reactive planning paradigm

The reactive planning paradigm, as shown in Fig. 3, is hierarchically organized by a collection of prioritized interaction pipelines of sensing-acting. Each interaction pipeline senses a specific environmental state and specifies a fast-responding action to handle the possible state changes. The paradigm maintains an arbitration scheme to prioritize these pipelines to express the significance of different state changes. The algorithmic description of the reactive planning paradigm is illustrated in Algorithm 2.

4.3.3 Hybrid planning paradigm

The hybrid planning paradigm, as shown in Fig. 4, combines the advantages of hierarchical and reactive paradigms by integrally maintaining two layers of sensing, planning, and acting interaction pipelines. The sensing functions formulate domain knowledge and real-time sensory events to the planning and acting functions, allowing for time-consuming deliberation on task plans and fast reactions to environmental changes. As long as the task plan has not finished execution, the set of sensing-acting interaction pipelines is kept running along with the task plan process and can be triggered according to pre-defined priorities. When some environment condition is detected as unsafe, the task plan will be terminated, and the corresponding sensing-acting pipeline outputs a reactive action for fast response based on user-defined reactive rules. The procedural algorithm of the hybrid paradigm can be jointly described based on the above two algorithmic descriptions in Algorithm 3.

4.3.4 Offline probabilistic planning paradigm

The offline probabilistic planning paradigm is similar to the hierarchical paradigm in performing one-shot planning for a complete task plan. However, the offline probabilistic task plan features a different contingent branched structure than the sequentially ordered structure of the hierarchical task plan, which contains both sensing and actuating action operators. The contingently branched task plan requires the periodic dispatching and feedback of each executed sensing and actuating action operator, whose plan execution results closely depend on the external environment states and the observations from sensing action operators. Based on these features, the algorithmic description of the offline probabilistic paradigm is shown in Algorithm 4.

4.3.5 Online probabilistic planning paradigm

Unlike the above paradigms that comprise the complete task plan, the online probabilistic planning paradigm performs periodic planning and outputs a one-shot action operator for each planning step. As shown in

Algorithm 1 Composition algorithm for the hierarchical planning paradigm

1: $f_s(I,a_s)=K$
2: $\Pi=\langle K,I,G\rangle$
3: $\kappa_{sp}: f_s \xrightarrow{\Pi} f_p, f_s \ominus f_p$
4: $f_p(\Pi)=\Delta$
5: $\kappa_{pa}: f_p \xrightarrow{\Delta} f_a, f_p \circledcirc f_a$

Algorithm 2 Composition algorithm for the reactive planning paradigm

1: $f_{s_0}(s_t,a_{s_0})=z_t, \mathrm{Rule}(s_t,z_t)=a_{a_0}, f_{a_0}(s_t,a_{a_0})$
2: $\kappa_{sa_0}: f_{s_0}\rightarrow f_{a_0}, f_{s_0}\circledcirc f_{a_0}$
3: $f_{s_1}(s_{t'},a_{s_1})=z_{t'}, \mathrm{Rule}(s_{t'},z_{t'})=a_{a_1}, f_{a_1}(s_{t'},a_{a_1})$
4: $\kappa_{sa_1}: f_{s_1}\rightarrow f_{a_1}, f_{s_1}\circledcirc f_{a_1}$
5:...
6: $\kappa_{sa_n}: f_{s_n}\rightarrow f_{a_n}, f_{s_n}\circledcirc f_{a_n}$
7: Prioritize$(\kappa_{sa_0},\kappa_{sa_1},...,\kappa_{sa_n})$

Algorithm 3 Composition algorithm for the hybrid planning paradigm

1: $f_s(I,a_s)=K$
2: $\Pi=\langle I,G,K\rangle$
3: $\kappa_{sp}: f_s \xrightarrow{\Pi} f_p, f_s \ominus f_p$
4: $f_p(\Pi)=\Delta$
5: $\kappa_{pa}: f_p \xrightarrow{\Delta} f_a, f_p \circledcirc f_a$
6: **while** κ_{pa} has not finished **do**
7: Parallelize $(\kappa_{sa_0},\kappa_{sa_1},...,\kappa_{sa_n})$
8: Prioritize $(\kappa_{sa_0},\kappa_{sa_1},...,\kappa_{sa_n})$
9: **if** some κ_{sa_i} has detected a state change **then**
10: Terminate $(f_a(\Delta))$
11: $\kappa_{sa_i}: f_{s_i}\rightarrow f_{a_i}, f_{s_i}\circledcirc f_{a_i}$
12: **end if**
13: **end while**

Algorithm 4 Composition algorithm for the offline probabilistic planning paradigm

1: $f_s(I,a_s)=K$
2: $\Pi=\langle I,G,K\rangle$
3: $\kappa_{sp}: f_s \xrightarrow{\Pi} f_p, f_s \ominus f_p$
4: $f_p(\Pi)=\Delta$
5: $\kappa_{pas}: f_p \xrightarrow{\Delta} f_a/f_s, f_p \circledcirc f_a/f_s$

Fig. 6, the sensing functions produce initial domain knowledge for the planning function to compute the first best action operator. After executing the action in the acting function, the sensing functions obtain a real-time observation and send it to the planning function for next-step planning. The interaction pipelines of sensing-planning-acting keep running periodically until the robot finally achieves the goal. The algorithmic description of this paradigm is in Algorithm 5.

5 Reusable Behavior Tree Structures for Interaction Pipelines

The previous section has proposed a set of standard interaction pipelines for synchronization and cooperation between the sensing, planning, and acting functions. The above composition algorithms show that the well-decomposed and abstracted interaction pipelines can be composed flexibly to constitute the different robot task planning paradigms. In this section, we take a step further to make these common interaction pipelines easily reusable in robotic software development. We utilize the essential components and interfaces of the Behavior Tree (BT) software development framework to develop a set of reusable behavior tree structures. Each BT structure can concretely implement the above abstract-level interaction pipeline. It provides well-defined interfaces for easy programming of inter-component communication and complex controlling and planning flows. The overview of our Structural-BT approach is shown in Fig. 1.

5.1 Prior knowledge of BT framework

The component-based BT software development framework has recently become increasingly popular in the robotic community. It dramatically improves robotic software development efficiency by introducing modular and reusable control logic in software components. A BT is a new way to structure the switching between different robot behaviors in autonomous robotic software. A BT is a directed rooted tree whose internal nodes are called the control flow nodes and external leaf nodes are referred to as the execution nodes. In the classical BT framework[19], there are four types of control flow nodes, including the Sequence, Fallback, Parallel, and Decorator nodes, and two execution nodes of Action and Condition. The advantage of BT components in robotic software development is the encapsulation of common controlling logic into reusable control flow nodes, making implementing controlling logic in robotic software easier than existing component-based approaches, such as the ROS framework. BT-based robot software starts its execution from the root BT node and generates execution signal ticks with a given frequency. The rest BT node components start to execute if and only if it receives ticks from its parent nodes. We discuss the reusable control flows of Sequence, Parallel, and Fallback nodes and two execution nodes, which are the foundation for our developed reusable BT structures. Figure 7 presents the graphical representations of these nodes.

For a Sequence node with N children (Fig. 7a), it routes ticks to its children from the left until it finds a child node that returns either Failure or Running. The Sequence node returns Success if and only if all its children return Success (Algorithm 6[19]). A Fallback node (Fig. 7b) routes ticks to its children from the left until it finds a child node that returns either Success or Running. It returns Failure if and only if all its children return Failure (Algorithm 7[19]). A Parallel node (Fig. 7c) routes the ticks to all the children nodes. It returns Success if M children nodes return Success, or

Algorithm 5 Composition algorithm of the online probabilistic planning paradigm

1: $f_s(I = s_t, a_s) = K$
2: $\Pi_t = \langle I, G, K\rangle$
3: $\kappa_{sp} : f_s \xrightarrow{\Pi_t} f_p, f_s \circledcirc f_p$
4: $f_p(\Pi_t) = \Delta = a_t^*$ {a single action operator}
5: **while** G has not been achieved **do**
6: $\kappa_{pa} : f_p \xrightarrow{a_t^*} f_a, f_p \ominus f_a$
7: $f_s(s_{t+1}, a_s) = z_{t+1}$
8: $\Pi_{t+1} = \Pi_t + z_{t+1}$
9: $f_p(\Pi_{t+1}) = a_{t+1}^*$
10: $\kappa_{sp} : f_s \xrightarrow{\Pi_{t+1}} f_p, f_s \circledcirc f_p$
11: **end while**

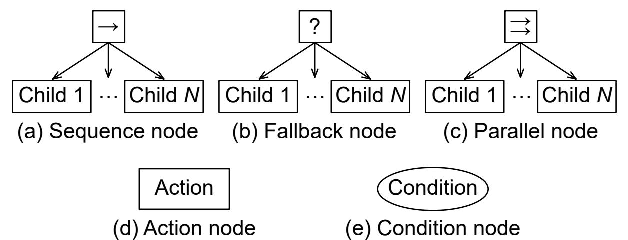


Fig. 7 Graphical representation of a Sequence control node (→), a Fallback control node (?), a Parallel control node (⇉), an Action node, and a Condition node.

Algorithm 6 Pseudocode of Sequence node with N children

```
1: for i ← 1 to N do
2:    childStatus = Tick(child(i));
3:    if childStatus == Running then
4:        return Running;
5:    else if childStatus == Failure then
6:        return Failure;
7:    end if
8: end for
9: return Success;
```

Algorithm 7 Pseudocode of Fallback node with N children

```
1: for i ← 1 to N do
2:    childStatus = Tick(child(i));
3:    if childStatus == Running then
4:        return Running;
5:    else if childStatus == Success then
6:        return Success;
7:    end if
8: end for
9: return Failure;
```

returns Failure if $N-M+1$ children nodes return Failure (Algorithm 8[19]). For an Action node, it starts to execute the encoded low-level action commands when it receives ticks. An Action node returns Success if the action execution is completed or Failure if the action has failed. While the Action node does not finish executing the commands, the node status returns Running. For a Condition node, it checks a proposition that describes a certain environment condition when it receives ticks. A Condition node returns Success or Failure depending on whether the proposition holds or not. In the classical BT framework, the proposition checking process is assumed to be instantaneous, and the Condition node never returns a status of Running[19].

The classical BT framework provides a modular

Algorithm 8 Pseudocode of Parallel node with N children

```
1: for i ← 1 to N do
2:    childStatus = Tick(child(i));
3: end for
4: if Sum(childStatus(i)==Success) ⩾ M then
5:    return Success;
6: else if Sum(childStatus(i)==Failure) > N−M then
7:    return Failure;
8: else
9:    return Running;
10: end if
```

design for robotic software. It is the first work to encapsulate the general-purpose controlling logic into reusable control flow nodes. The modular design and encapsulation of controlling flow significantly reduce the programming efforts required for complex controlling logic implementation, compared with the limited reusability of data communication pipelines in the ROS component framework. However, the classical BT framework and other popular component-based robotic software development frameworks provide no encapsulation and reusable designs regarding task planning paradigms' more cumbersome implementation issue, which generally requires more programming efforts. In this work, we utilize some of the standard concepts of BT and develop a set of reusable BT structures that could facilitate the efficient implementation of abstract task planning paradigms.

5.2 Domain-specific robotic BT components

We first develop a set of domain-specific BT software components that closely relate to the task planning functionality in robotic software. We have designed concrete component models and reusable interfaces for the critical computation functions of sensing, planning, and acting. The general computation scheme can be flexibly reused in each component, and software engineers can easily customize the task-specific settings of different planners, sensors, and actuators.

Definition 12 (Sensing action node) The sensing action node $\mathcal{N}_s$ inherits the basic tick engine from the BT action node that waits for external ticks for execution, and also incorporates a task-specific sensor that senses environment states for sensory information and formulates high-level observations or generates the task-specific problem model. Formally, the main model elements of a sensing action node can be described as $\mathcal{N}_s=\langle\mathcal{N},\alpha\rangle$, where the $\mathcal{N}$ represents the basic BT action node that implements the tick engine, and α represents a robotic sensor component that interacts with external environments and receives sensory information. Figure 8 graphically describes the internal component elements of a sensing action node. The $\mathcal{N}_s$ component could receive either raw sensory information (such as the image and laser readings) or a domain model (the contextual information of specific tasks) and output a high-level observation (formulated knowledge based on raw sensory information) or a problem model. The internal sensor component provides two methods of ObservationFormulation() and RunProblemGenerator() that could be user-

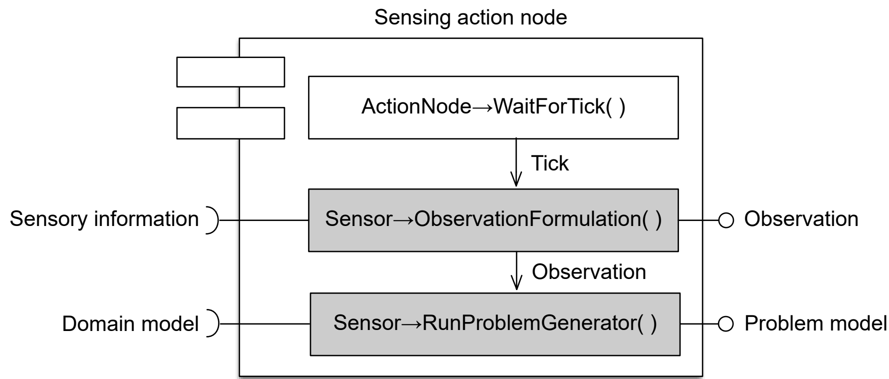


Fig. 8 Graphical representation of the BT-based sensing action node component.

customized for different task requirements.

Definition 13 (Planning action node) The planning action node $N_{\mathrm{p}}= \langle N, \beta \rangle$ also inherits the basic tick engine from the BT action node $\mathcal{N}$ and incorporates a task-specific planner component β that performs diverse types of task planning, such as the POPF planner☆ for hierarchical planning and the DESPOT planner† for online probabilistic POMDP planning. Figure 9 shows the internal model elements of the BT-based planning action node. The component $\mathcal{N}_p$ generally provides interfaces for receiving a problem model and outputting a generated plan. The planner component provides the RunPlanning() method for customizing internal planning algorithms based on diverse task requirements.

Definition 14 (Acting action node) The acting action node $\mathcal{N}_a$ is dedicated to executing a task-level plan by implementing the motion-based actor for concrete acting with low-level robotic hardware actuators. The $\mathcal{N}_a$ component consists of the basic BT action node $\mathcal{N}$ for receiving the ticks and starting the motion-based acting process and the actor γ for implementing the specific motion-based acting process. For example, when receiving the task-level plan of goto (pos1, pos2), a move base actor γ, which interfaces with the hardware move base actuator and provides the Proportional-Integral-Derivative (PID) controlling motion algorithm, could concretely execute the plan and drive the robot to reach the goal destination in the real-world environment. As shown in Fig. 10, the actor component receives the plan and maintains the customizable function RunConcreteActing() for different actuators.

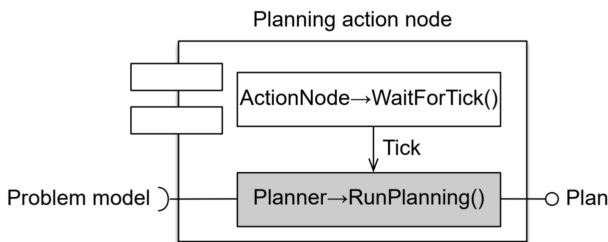


Fig. 9 Graphical representation of the BT-based planning action node component.

☆https://github.com/fmrico/popf.git

†https://github.com/AdaCompNUS/despot.git

5.3 BT structure realization of interaction pipeline

Based on the above robot-specific functional BT components, we develop a set of reusable BT structures to implement the abstract interaction pipelines. The reusable BT structures can then be flexibly composed to quickly develop the skeleton architecture of robotic software, which improves the development efficiency.

5.3.1 Reusable BT structure for S-P interaction pipeline

Figure 11 presents external and internal views of the BT structure for the sensing-planning interaction pipeline. As shown in the external view (Fig. 11a), the BT structure $\mathcal{T}_{sp}$ uses the sequence node as its root node and adds sensing $\mathcal{N}_s$ and planning $\mathcal{N}_p$ action nodes as the children nodes. In the structure, the controlling and planning flow in the interaction pipeline are implemented synchronously and could be encapsulated within the structure for software reuse. As

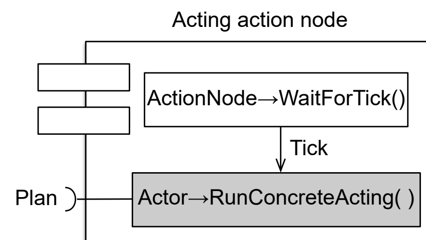


Fig. 10 Graphical representation of the BT-based acting action node component.

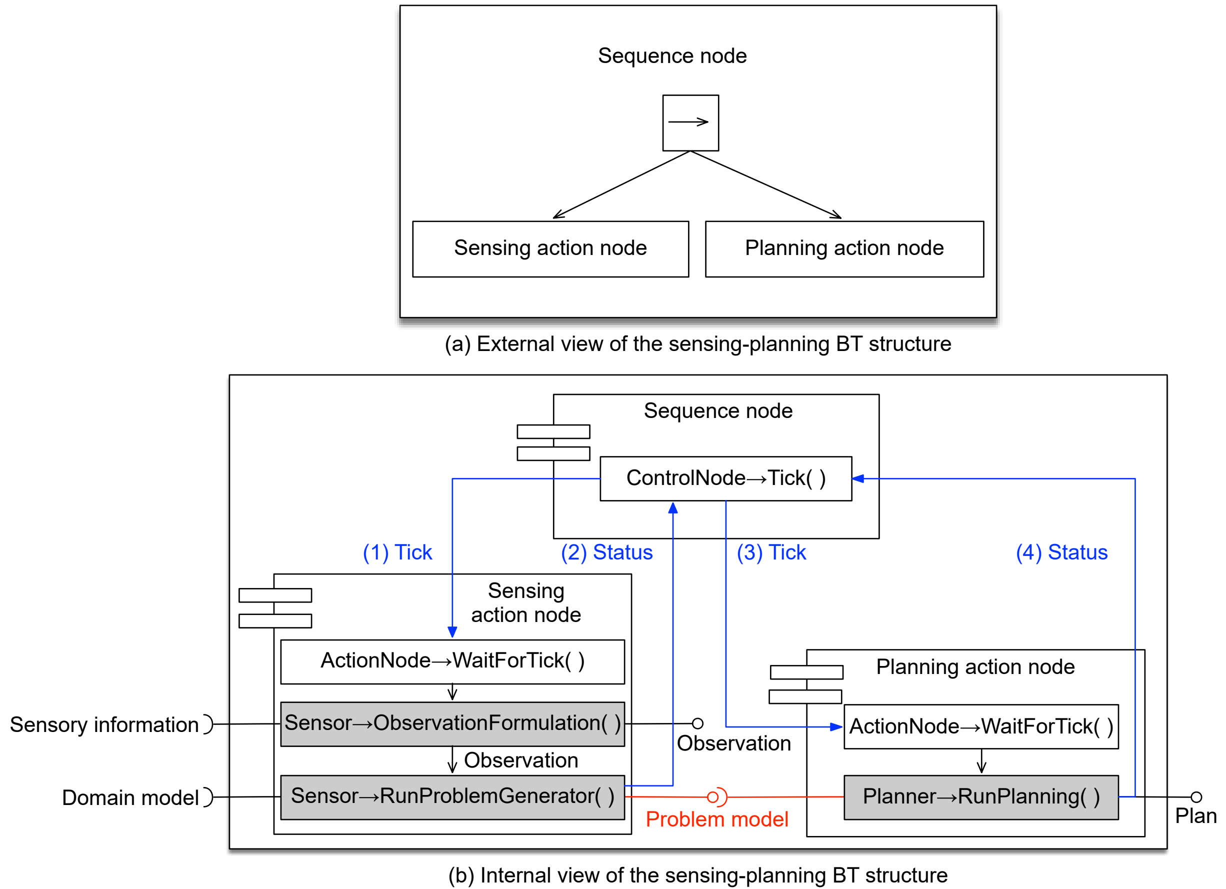


Fig. 11 Graphical representation of (a) external and (b) internal views of the BT structure for the implementation of sensing-planning interaction pipeline.

shown in Fig. 11b, the root sequence node routes the execution signal tick to the sensing and planning action nodes sequentially: (1) the root node sends the tick to the sensing node $\mathcal{N}_s$, ticking the sensor to formulate an observation and generate a problem model; (2) after successfully generating the problem model, the $\mathcal{N}_s$ returns its execution status of "success" back to the root node; (3) the root node then sequentially routes the tick to the planning node $\mathcal{N}_p$ to trigger the planner to perform task planning; (4) as soon as the task plan is computed, the $\mathcal{N}_p$ node returns the status of "success" back to the root node, and the root node subsequently set the status of this structure as "success". Notably, the above controlling flow of tick and status routing is implemented in sync with the planning flow of problem model generation and task plan computation, making this BT structure a well-behaved and independent composite component for software reuse.

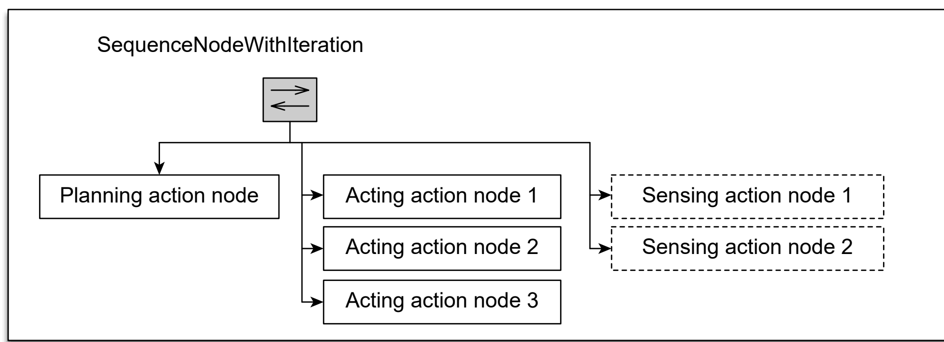
5.3.2 Reusable BT structure for P-A/S interaction pipeline

The aforementioned planning-acting/sensing interaction pipelines are dedicated to dispatching and executing a task plan that contains an ordered set of actuating/sensing action operators. We develop a BT structure that is capable of implementing both the planning-acting κ_{pa} and planning-acting/sensing κ_{pas} interaction pipelines. As κ_{pa} and κ_{pas} implement a periodic plan dispatching process, we develop a new BT control node of SequenceNodeWithIteration to realize the iterative control logic. In the node, we implement an iterative procedure of "action selection−action dispatch−action execution−action feedback" inside the function of "ControlNode→Tick()". With each tick arriving at the SequenceNodeWithIteration node, the iterative procedure will be executed once, along with routing a tick to the children's action nodes. In this case, the iterative procedure of plan execution is kept synchronous with the control flow of tick routing. Notably, to enable easy composition of the S-P and P-A/S structures, we add the interface of SequenceNodeWithIteration node to receive the task

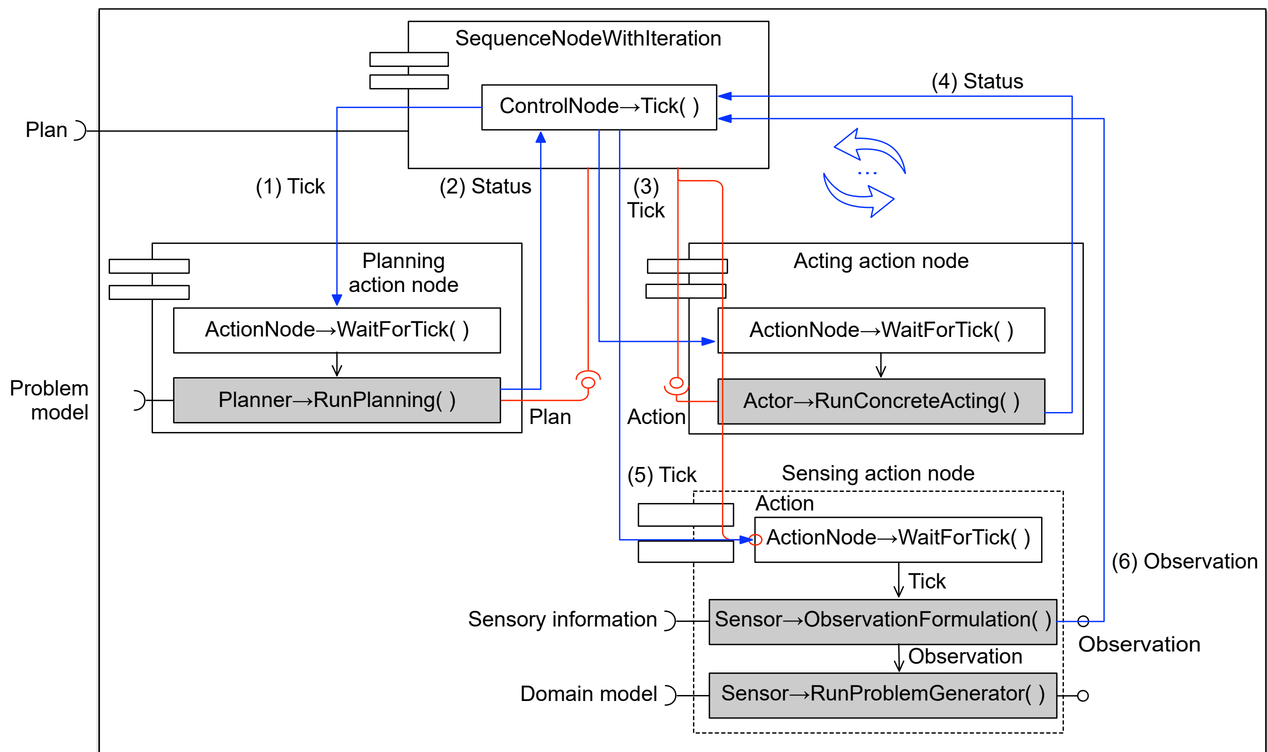
plan from external data flow to avoid the redundancy of planning node implementation.

Figure 12 presents external and internal views of the reusable BT structure for the planning-acting/sensing interaction pipeline. As shown in the external view (Fig. 12a), the SequenceNodeWithIteration node is set as the root node. In the BT structure, we add a planning action node $\mathcal{N}_p$, a set of acting action nodes $\{\mathcal{N}_{a_1}, \mathcal{N}_{a_2}, \ldots, \mathcal{N}_{a_n}\}$, and possibly a set of sensing action nodes $\{\mathcal{N}_{s_1}, \mathcal{N}_{s_2}, \ldots, \mathcal{N}_{s_n}\}$ as the children nodes that handle the different actuating/sensing action operators of the deterministic/non-deterministic task plans. The BT structure can support the planning-acting (P-A) interaction pipeline by dispatching each actuating action operator of the deterministic task plan to the acting action nodes. It can support the planning-acting/sensing (P-A/S) interaction pipeline by dispatching each actuating or sensing action operator of the non-deterministic task plan to the corresponding acting or sensing action nodes.

As shown in Fig. 12b, the iteration scheme of action dispatching and feedback is implemented in sync with the controlling flow of tick routing. (1) The root node sends the tick to the planning node $\mathcal{N}_p$ and triggers the planner to make up the task plan. (2) As soon as the planning is finished, the $\mathcal{N}_p$ returns the status of "success" back and the generated task plan to the root node. The root node then starts the iterative procedure



(a) External view of the planning-acting/sensing BT structure



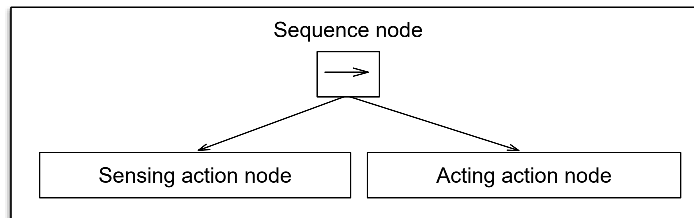
(b) Internal view of the planning-acting/sensing BT structure

Fig. 12 Graphical representation of (a) external and (b) internal views of the BT structure for implementation of the planning-acting/sensing interaction pipeline.

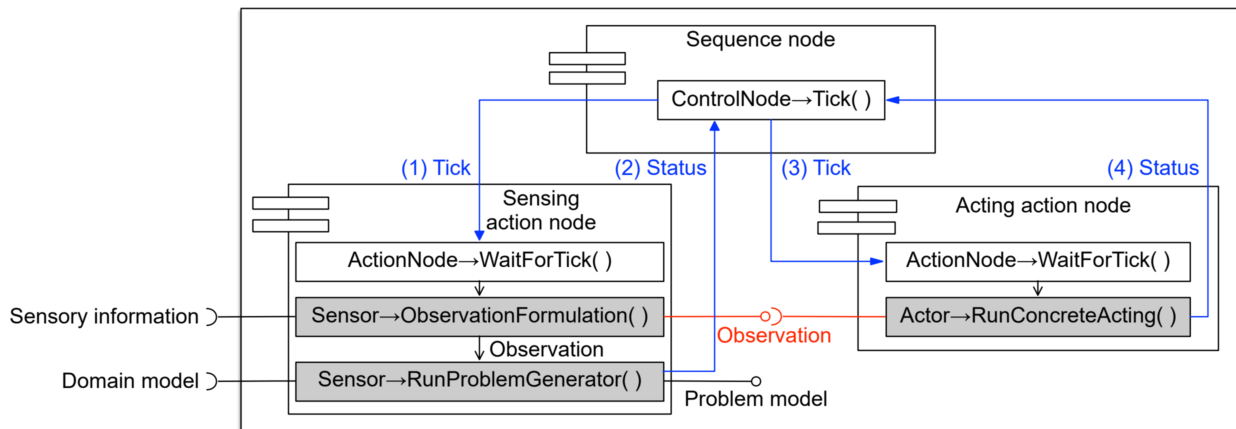
and dispatches each action operator from the task plan to the corresponding (3) acting (5) sensing action node. The actuating/sensing action node starts its execution by receiving the action operator and execution tick. The corresponding acting node $\mathcal{N}_{a_i}/\mathcal{N}_{s_i}$ finishes executing the action and returns the execution (4) status/(6) observation back to the root node. Notably, the execution feedback of a sensing action node is the received observation, whereas the actuating action node returns the execution status of "success", "failure", or "running". Both types of action feedback can be processed in the same fashion by the iterative procedure of the SequenceNodeWithIteration node. The root node could repeat the steps of (3) and (4) or (5) and (6) by iteratively dispatching the actuating or sensing action operators to the acting or sensing function. The root node maintains the iterative dispatching and acting loop until the task plan has executed all its action operators.

5.3.3 Reusable BT structure for S-A interaction pipeline

The interaction pipeline between the sensing and acting functions, which avoids a deliberative planning process, is dedicated to reactively responding to runtime environmental changes. The reactive planning paradigm is essentially a prioritized hierarchy of multiple interaction pipelines between the sensing and acting functions. In each interaction pipeline, the sensing function checks a condition proposition with the sensory information, and the acting function runs the corresponding action when the condition is unsatisfied. To implement the pipeline, we develop the sequence tree structure $\mathcal{T}_{sa}$ and encapsulate the controlling and data flows inside the structure, enabling the reuse of sensing-acting interaction logic in robotic software development. Figure 13a graphically depicts an external view of the BT structure, and Fig. 13b describes the controlling and data flows in the structure. (1) In the structure $\mathcal{T}_{sa}$, the sequence root node first routes the tick to the sensing action node $\mathcal{N}_s$ and triggers the sensor to formulate an observation based on sensory information. The observation is then utilized to check whether some environmental condition is satisfied. (2) If the condition is satisfied, the sensing node $\mathcal{N}_s$ then returns the status "success" back to the root node. (3) The root node then routes the tick to the acting action node $\mathcal{N}_a$. It triggers the actor to execute the corresponding reaction action operator (specified by user-defined reactive rules). (4) The acting node $\mathcal{N}_a$ waits for the execution of the action operator and returns the execution status to the root node.



(a) External view of the sensing-acting BT structure



(b) Internal view of the sensing-acting BT structure

Fig. 13 Graphical representation of (a) external and (b) internal views of the BT structure for implementation of the sensing-acting interaction pipeline.

6 Experiment

We select four robotic tasks with diverse goals and environmental conditions to validate the software development efficiency of our reusable BT structures. We then develop software systems for each task to measure the level of development effort quantitively. The selected robotic tasks could be actual samples to evaluate the reusability of each abstracted interaction pipeline and concrete BT structures. Moreover, to make the results as objective as possible, we select two of the most popular component-based robotic software development frameworks (ROS and classical BT) as the baseline approaches. We show through this comparison that the Structural BT approach has improved reusability compared to existing approaches. The source codes of our Structural-BT component framework and the software implementation for the experimental tasks can be available upon request.

6.1 Task scenarios

In this experiment, we have selected four diverse types of robotic domains and designed the task requirements with different environmental constraints. Below task scenarios are carefully selected and designed based on the following principles. **Firstly**, the task domains are relatively common in the robotic community, which receives many software development requirements. **Secondly**, the environmental constraints of four task domains have covered the uncertainties of dynamics and perception, such as Task 1 with the static and fully observable environment, Task 3 with the static and unknown environment, and Tasks 2 and 4 with the dynamic and unknown environment. **Thirdly**, these task domains generally consist of composite robotic actions and complex deliberation requirements, making developing corresponding software systems rather tricky. Testing our approach within these task scenarios could practically evaluate the applicability and effectiveness of the framework in the most common domains and complex environment constraints. As the hybrid paradigm is essentially an integral combination of hierarchical and reactive structures, the selected robotic task scenarios are designed to cover, in particular, the mainstream hierarchical, reactive, offline, and online probabilistic task planning paradigms. Figure 14 presents snapshots of the four task scenarios. The task descriptions are as follows.

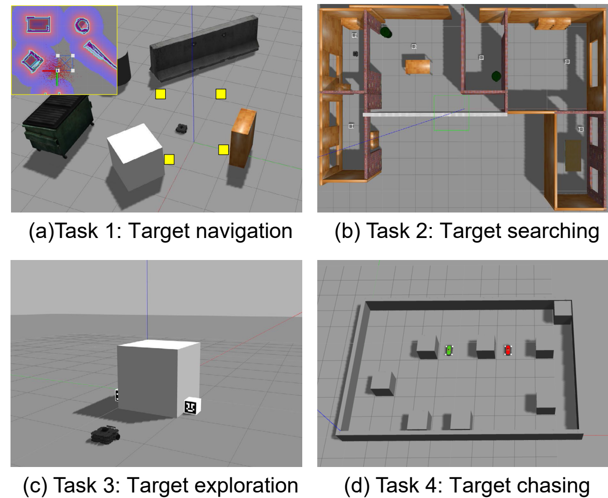


Fig. 14 Task scenarios of four indoor robotic tasks in the experiment.

- **Task 1 (target navigation):** The robot is expected to navigate in an indoor environment to visit a set of target destinations. In this task, the robot can initially obtain positions and target destinations with sensory information from robotic sensors. Then the robot needs to perform hierarchical task planning based on the known information to compute the task plan that minimizes movement cost.
- **Task 2 (target searching):** The robot is expected to search for a target object that remains unknown in a dynamic environment. In the environment, moving obstacles may appear in the way and cause possible collisions. For safety, the robot needs to prioritize detecting and avoiding moving obstacles, then searching for the target object by randomly wandering in the room. The software needs to perform reactive planning that achieves the reactive goal of obstacle avoidance.
- **Task 3 (target exploration):** The robot is expected to explore two unknown areas to find a target object. In the task, the robot can partially estimate the target's existence by obtaining a partial view image, which causes uncertainties in target recognition. The robot must make a complete task plan containing necessary sensing actions for checking environment states and actuating actions for exploring the environment. The software is expected to perform offline probabilistic task planning.
- **Task 4 (target chasing):** The robot is expected to chase after a moving target in a partially observable environment. The robot can roughly obtain the target's direction by continuously obtaining and analyzing laser readings. As the target keeps moving, the robot must perform online probabilistic planning to decide the best movement action. After each action execution, the

robot can get closer to the target. The online planning process continues until the robot reaches the target's position.

6.2 Baseline approach

In this experiment, we have selected the ROS‡[16] and classical BT¤ component frameworks as the baseline approaches of robotic software development. Both frameworks have been popular in robotics and software engineering communities due to their well-designed component interfaces, functionality encapsulation, and modular design. With these frameworks, robotic engineers can easily reuse remote communication infrastructures (the topic/service schemes in ROS) and versatile controlling logic (such as the sequence/fallback/parallel control flows in classical BT) when developing software for different robotic tasks.

In Fig. 15a, the ROS framework includes the ROS node component that encapsulates the remote data communication channels (topic/service) as reusable infrastructure. Software engineers can easily reuse the topic/service invocation interfaces when architecting communications between functional software components. In Fig. 15b, the classical BT framework has provided a set of BT components with versatile types, including control and execution nodes. BT control nodes have improved the reusability level over the ROS framework by creatively encapsulating the basic controlling logic of sequence, fallback, and parallel, which are commonly required and implemented in robotic software development. The interfaces of BT control nodes and action/condition nodes are modular and reusable, which relieves the software engineers from manually implementing the basic controlling flows between robotic software

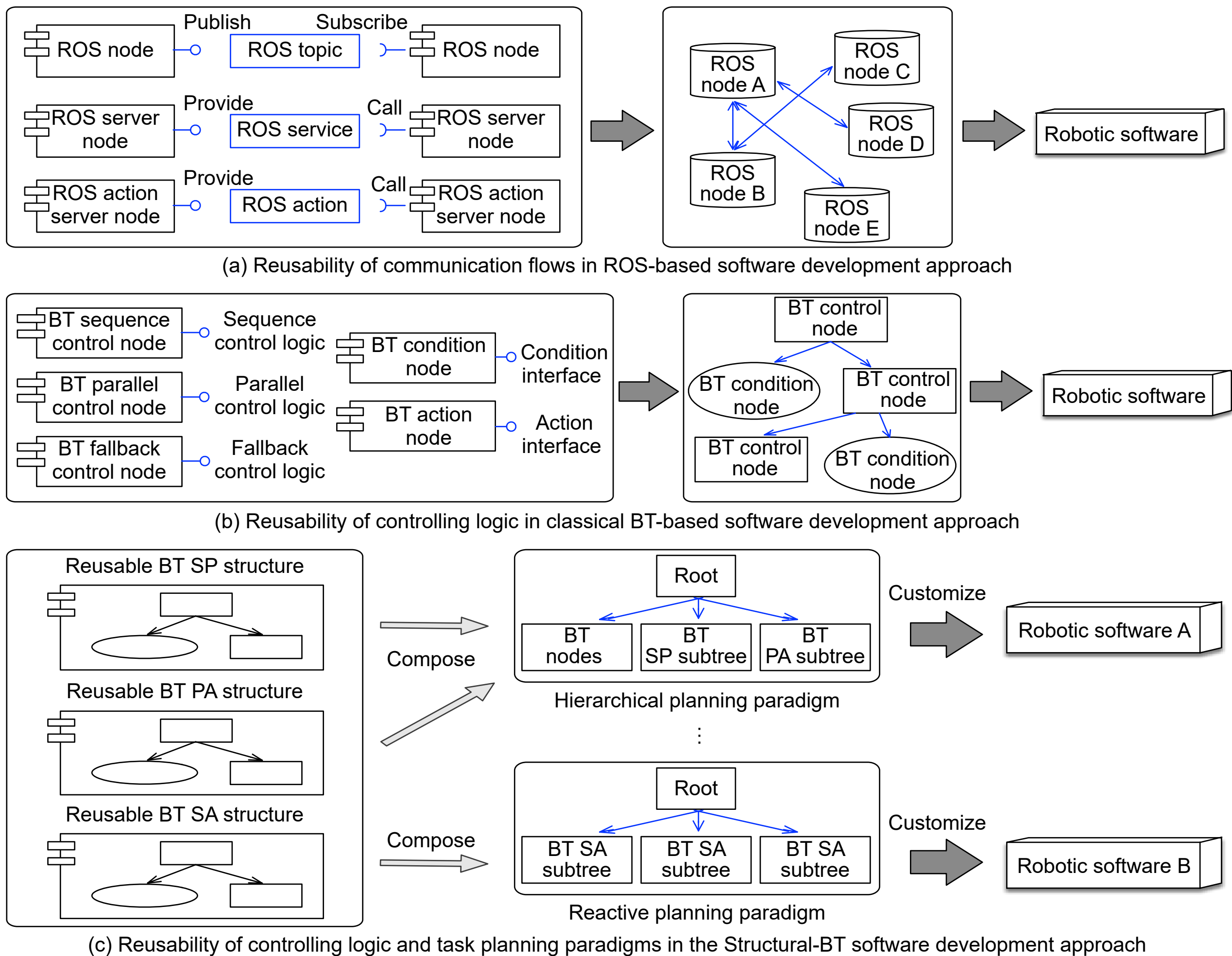


Fig. 15 Different reusability levels of the ROS, classical BT, and Structural-BT development approaches.

‡http://wiki.ros.org/kinetic/Installation/Ubuntu
¤https://github.com/miccol/ROS-Behavior-Tree.git

components. Figure 15c graphically presents the reusable design of our proposed approach. Taking it a step further, our approach proposes to minimize the complexity of robotic software development by reusing the basic and common patterns of interaction pipelines between the sensing, planning, and acting functions. The set of reusable BT structures corresponding to the interaction pipeline patterns can be flexibly composed to implement a specific task planning paradigm and formulate the skeleton software architecture. Software engineers only need to customize the task-specific planners and hardware-specific sensors/actors inside the software without much programming effort to implement the basic controlling flows and task planning paradigms.

Figure 16 presents the representative process of composing an S-P structure and P-A/S structure for developing software for Task 3. The software of Task 3 requires the implementation of an offline probabilistic task planning paradigm. We select the reusable S-P and P-A/S structures to compose a skeleton BT tree that establishes the task planning and plan execution framework. The planning action node has been implemented and incorporated into the S-P structure. The node outputs the non-deterministic task plan and sends it to the receiver interface of the adjacent subtree of the P-A/S structure via the inter-node mutual data sharing scheme. The mutual data sharing of BT nodes has been previously studied in our earlier work[23], which can be referred to for more details.

6.3 Result and analysis

In this experiment, we quantitatively evaluate the reusability level of components from three comparative approaches by measuring their reuse frequency and reuse ratio. The reuse frequency of a component shows the extent to which the component is selected for reuse in other components or projects by developers. A high reuse frequency of a component means this component could be rather beneficial and demanding in various robotic software projects. The reuse ratio refers to the ratio of the reused amount of code in a task-specific software component. In a software component built by developers, a high reuse ratio means the majority of lines of code of this component could be reused from existing code repositories, reducing the developers' programming effort.

We first measure the reuse frequency of the ROS components (ROS approach), BT components (classical BT approach), and BT structures (Structural-BT approach) to show their applicability in robotic software development. Then we take it further to evaluate the reuse ratio of code amounts of the above components or structures in each application-specific component.

6.3.1 Reuse frequency

Figure 17 presents the reuse frequencies of the diverse components or structures provided by the ROS,

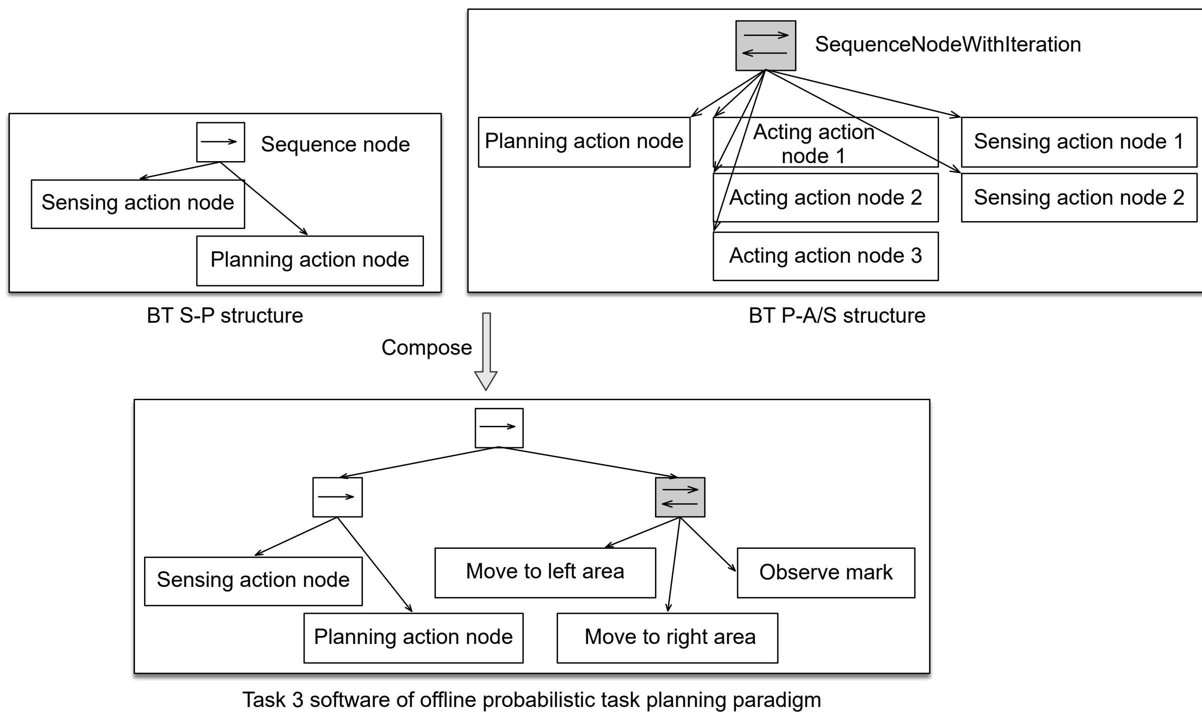
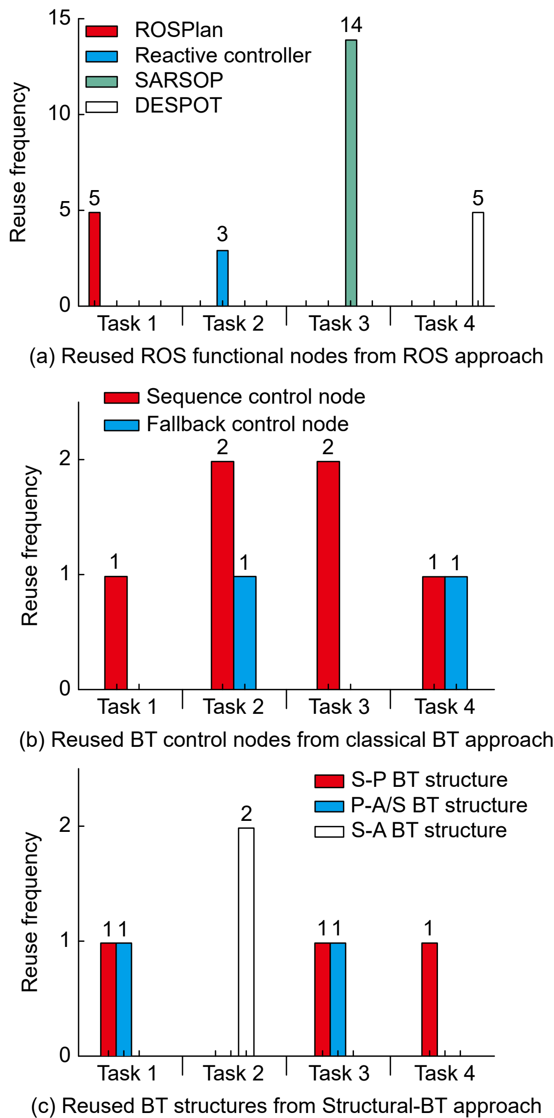


Fig. 16 Process of composing the reusable S-P and P-A/S structures to develop the Task 3 software.



(a) Reused ROS functional nodes from ROS approach

(b) Reused BT control nodes from classical BT approach

(c) Reused BT structures from Structural-BT approach

Fig. 17 Reuse frequencies of the diverse components or structures provided by three comparative approaches for the four task scenarios.

classical BT, and our proposed Structural-BT development approaches. The reuse frequency of a reusable component or structure is measured by the number of times it is invoked in each task-specific robotic software instance.

Figure 17a shows the reuse frequency of four task-related ROS functional nodes among the four task scenarios. In the ROS development approach, Tasks 1−4 require different task planning paradigms and more detailed algorithmic implementation of task-specific planners, sensors, and actuators. In this approach, we utilize the core ROS functional nodes of the ROSPlan¶ POPF planning framework to design and develop the robotic software for the Task 1 scenario. Within this context, we refer to a class or package's source code that implements an independent functionality as a functional component. Five core functional ROS components are reused in the robotic software for Task 1. Similarly, we utilize the off-the-shelf ROS functional nodes from the reactive controller, SARSOP¥ offline probabilistic planning framework, and DESPOT© online probabilistic planning frameworks to develop the robotic software for Tasks 2−4, respectively. Each ROS-based planning framework offers highly specialized and technically diverse planning algorithms in the software development process, which can effectively solve the corresponding task planning problems and efficiently implement the software by reusing some planning components. However, none of these frameworks, such as ROSPlan and SARSOP, can be well reused among different tasks due to the lack of reuse in the common and general controlling and planning flows. In Fig. 17a, we find out that each ROS-based planning framework can only be applicable in their corresponding task scenario but fail to be reused in other task scenarios requiring different task planning paradigms.

The classical BT framework enables the reuse of common controlling logic, such as sequential, fallback, and parallel control loops, by encapsulating them into the Sequence, Fallback, and Parallel control nodes, which makes them fully reusable without any code modification. Moreover, the classical BT framework provides a unified modeling and programming framework that unifies the C++/Python programming languages and generalized component interfaces, making it widely applicable to robotic software programming. In Fig. 17b, the Sequence control node has been reused multiple times in all the tasks that serve as the primary sequential controlling loops. The Fallback control node has also been reused in Tasks 2 and 4, which implement the primary if-else selection controlling loops.

Figure 17c records the reuse frequencies of the three proposed BT structures that specifically implement the interaction pipelines between sensing, planning, and acting in our Structural-BT development approach. The BT structures of S-P, P-A/S, and S-A implement the commonly used planning and controlling flows that

¶https://github.com/KCL-Planning/ROSPlan.git

¥https://github.com/AdaCompNUS/sarsop.git

©https://github.com/AdaCompNUS/despot.git

explicitly exist in the above interaction pipelines. The BT structures have been designed to be modular, general, and customizable, enabling the encapsulated planning and controlling flows widely applicable in most robotic tasks. In Fig. 17c, the S-P BT structure has the highest reuse frequency in task scenarios 1, 3, and 4. These tasks require a deliberative planning scheme that first senses the environment to formulate domain information and then performs diverse planning algorithms to make a valid task plan. Similarly, the P-A/S BT structure is reused in Tasks 1 and 3 as the output task plans generally consist of action operators, which require developing the iterative loops of plan dispatching and feedback for the underlying software. The S-A BT structure is reused twice in Task 2, which couples the direct data and control flows between a robotic sensor and an actuator. Software developers could reuse this structure multiple times to develop the hierarchically organized layers in the reactive planning paradigm.

6.3.2 Reuse ratio

The above statistics reveal the general reusability levels of the BT/ROS components or BT structures from three comparative approaches, demonstrating the applicability of these models by comparing their reuse frequencies in diverse types of robotic tasks. We further examine the specific reusability ratio of each reused component or structure concerning the code size, which indicates the programming effort required for robotic software development.

In the ROS approach, we develop the software for different tasks by inheriting ROS node components' basic interfaces and topic/service communication channels. We also implement concrete planning algorithms using technically different planner projects. The ROS node interfaces only provide the skeleton of remote data communication, while the algorithmic implementation of planning and controlling flows is fully task-customized. Most codes on the planning algorithm implementation of the ROSPlan, Reactive controller, SARSOP, and DESPOT are not reusable in different tasks, with only the ROS-related codes commonly reused among the above four tasks. Figure 18a depicts the reuse ratio of the ROS-related classes and functions (such as ros::ServiceClient and ros::Subscriber) in the four tasks. As shown, the ROS-related codes take the proportion of the total code size of software implementation for Tasks 1–4 at 23.8%, 16.3%, 16.9%, and 17.2%, respectively.

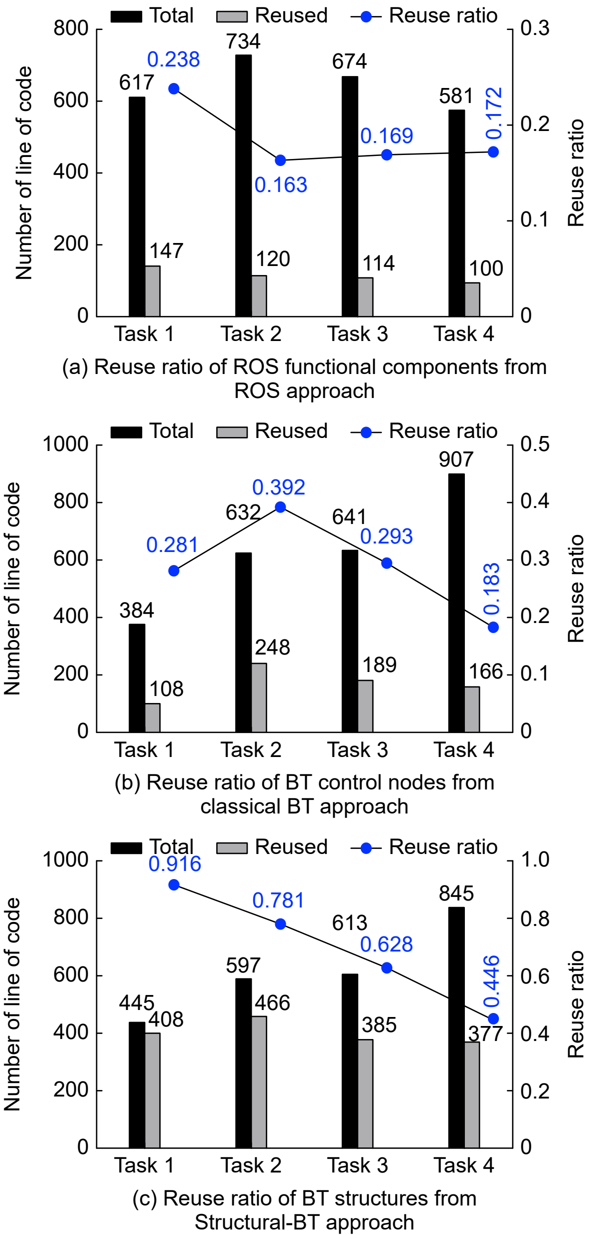


Fig. 18 Number of line of code and reuse ratios of the diverse components or structures provided by three comparative approaches for the four task scenarios.

The BT control nodes have been shown to have wide reusability for different task scenarios in the classical BT approach, with the encapsulated Sequence and Fallback controlling logic being fully reused in software development. We quantitatively measure the code sizes of the overall implementation of each task's software and the code sizes of utilized BT control nodes. The reuse ratios of BT control nodes for the four tasks are shown in Fig. 18b, which take the proportion of the total code size at 28.1%, 39.2%, 29.3%, and 18.3%. In our proposed Structural-BT approach, we

improve the reusability level of the controlling logic in classical BT to the integral interaction pipelines of planning, controlling, and data flows. The BT structures for S-P, P-A/S, and S-A interaction pipelines are implemented as independent and composite components, which formulate the program skeletons of the deliberative sensing-planning process, the iterative planning-acting process for plan execution, and the reactive sensing-acting process. Figure 18c shows the proportions of the total code sizes of S-P, P-A/S, and S-A BT structures concerning the total implementation size of each task's software. The statistics show improved reusability ratios of these BT structures for Tasks 1−4 at 91.6%, 78.1%, 62.8%, and 44.6%, respectively. Notably, the reusability ratios of the above components or BT structures closely depend on implementing other technical algorithms for sensor data processing, state-related searching strategies, and motion-level planning and acting. In the experiment, the Task 1 scenario is assumed to be a static and fully known environment, making hierarchical task planning relatively easier to implement and leading to a straightforward non-reusable technical algorithm implementation. The Task 4 scenario is dynamic, partially observable, and requires the more complex online probabilistic planning paradigm. The non-reusable technical algorithms for laser reading analysis and movement control make the total implementation code size relatively more significant than the software for other tasks. For this reason, the reusability ratios are decreasing from Tasks 1 to 4 due to the diverse implementation complexities. However, the reusability ratios of BT structures in the Structural-BT approach are generally higher than in the other two approaches.

6.4 Summary

To sum up, the ROS approach has the lowest reuse frequencies of its third-party functional components (such as ROSPlan and SARSOP) among diverse robotic tasks, with only essential ROS Topic/Service (T/S) communication infrastructures commonly reused. The average reused ratios of the ROS communication infrastructures reach 18.6% in our four task scenarios, which is the least ratio among the three approaches. The classical BT approach has a relatively higher reusability performance as it provides the commonly reusable Sequence (Seq) and Fallback (Fal) control nodes. The average reuse ratios of classical BT components (Seq and Fal) have reached 28.7%. The Structural-BT approach, which provides the reusable structures of S-P, P-A/S, and S-A, has reached the highest reuse ratios of 69.3%. This is because these reusable structures implement the common controlling logic and decision-making flows, not only the controlling logic, in the classical BT approach. The developers can reuse most lines of codes of these structures in their projects, which could greatly improve software development efficiency. The detailed comparison statistics have shown in Table 1.

7 Conclusion

This paper focuses on the issue of developing robotic software in open-world environments. The existing component-based development frameworks have limited reusability to the data and controlling flows without considering more complex task planning paradigms and underlying software architectures. This paper presents a novel component-based framework Structural-BT to provide a set of reusable BT structures, which implements a set of patterns for the sensing, planning, and acting interaction pipelines while providing interfaces for task-specific customization. This paper has developed the software for a set of tasks in an experiment using three comparative development approaches, including the ROS, classical BT, and our proposed Structural-BT approaches. The statistics show that the BT structures were generally reused more frequently than the components from other approaches. The reused code

Table 1 Comparison of reusability performances for three robotic software development approaches.

Task	ROS			Classical BT			Structural-BT		
	Reused component	LOC reuse ratio	Average LOC reuse ratio	Reused component	LOC reuse ratio	Average LOC reuse ratio	Reused component	LOC reuse ratio	Average LOC reuse ratio
1	T/S	0.238	0.186	Seq	0.281	0.287	S-P, P-A/S	0.916	0.693
2	T/S	0.163		Seq, Fal	0.392		S-A	0.781	
3	T/S	0.169		Seq	0.293		S-P, P-A/S	0.628	
4	T/S	0.172		Seq, Fal	0.183		S-P	0.446	

Note: LOC is the line of code.

sizes of BT structures have an average proportion of 69.3% of the full software implementation across software for four tasks in the Structural-BT approach. In contrast, the average reuse ratios of ROS or BT components are 18.6% and 28.7% in the ROS and classical BT approaches, respectively. The experiment has demonstrated the improved development efficiency of our proposed Structural BT approach for programming robotic software.

8 Future Work

While the robotic software of four task scenarios in the experiment was developed by the Structural-BT framework with a relatively high reuse performance, there are still several limitations in our preliminary research work, which need to be enhanced in future works. Firstly, the selected task planning paradigms may be inadequate for a wider range of robotic tasks. In this paper, we have comprehensively surveyed the research papers and robotic software projects in robotics and software engineering, synthesizing and identifying the five mainstream paradigms. The selected paradigms have shown to be applicable in the single robot tasks with simple action synchronization requirements. However, for more complex robotic domains that involve multi-robot cooperation, multi-task synchronization, etc., more types of task planning paradigms need to be analyzed and abstracted to cover more versatile robotic tasks. Secondly, some composable interfaces of behavior tree structures cause need extra programming effort when composing. When composing two reusable behavior tree structures, there may be a functional behavior tree node existing in the two structures, which needs tailoring the overall structure and upper control flow to resolve the conflict. In the future work, the composition flexibility of some repetitively embedded functional nodes needs to be enhanced by optimizing their model interfaces. Thirdly, the confidence level of experimental validation needs to be improved. This paper selects four typical robotic task scenarios and designs the simulation environment for experiments. The simulation experiments are conducted on the Gazebo robotic simulator, which has been widely recognized as the most popular and credible experimental platform. The robotic software that tested through Gazebo simulator can be easily transferred to the real robotic hardware without much modification. However, to make the validation results as convincing as possible, we would like to test the robotic software development efficiency of Structural-BT framework on more versatile types of robotic tasks and implement the software on real robotic hardware for more reliable statistics.

References

[1] H. S. D. Andrade, J. Schroeder, and I. Crnkovic, Software deployment on heterogeneous platforms: A systematic mapping study, *IEEE Trans. Softw. Eng.*, vol. 47, no. 8, pp. 1683–1707, 2021.

[2] D. Brugali, *Software Engineering for Experimental Robotics*. Heidelberg, Germany: Springer Berlin, 2007.

[3] D. Brugali and E. Prassler, Software engineering for robotics [from the guest editors], *IEEE Robotics Autom. Mag.*, vol. 16, no. 1, pp. 9–15, 2009.

[4] A. Shahbazian, Y. K. Lee, Y. Brun, and N. Medvidovic, Making well-informed software design decisions, in *Proc. 40th Int. Conf. Software Engineering: Companion Proceeedings*, Gothenburg, Sweden, 2018, pp. 262–263.

[5] E. Bouwers, A. V. Deursen, and J. Visser, Quantifying the encapsulation of implemented software architectures, in *Proc. 2014 IEEE Int. Conf. Software Maintenance and Evolution*, Victoria, Canada, 2014, pp. 211–220.

[6] Y. Tang, L. Li, and X. Liu, State-of-the-art development of complex systems and their simulation methods, *Complex System Modeling Simulation*, vol. 1, no. 4, pp. 271–290, 2021.

[7] P. Kruchten, R. Capilla, and J. C. Dueñas, The decision view's role in software architecture practice, *IEEE Softw.*, vol. 26, no. 2, pp. 36–42, 2009.

[8] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, Specification patterns for robotic missions, *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2208–2224, 2021.

[9] F. Ingrand and M. Ghallab, Deliberation for autonomous robots: A survey, *Artif. Intell.*, vol. 247, pp. 10–44, 2017.

[10] A. Chella, M. Cossentino, S. Gaglio, L. Sabatucci, and V. Seidita, Agent-oriented software patterns for rapid and affordable robot programming, *J. Syst. Softw.*, vol. 83, no. 4, pp. 557–573, 2010.

[11] D. Brugali and P. Scandurra, Component-based robotic engineering (part I) [tutorial], *IEEE Robotics Autom. Mag.*, vol. 16, no. 4, pp. 84–96, 2009.

[12] D. Brugali and A. Shakhimardanov, Component-based robotic engineering (part II), *IEEE Robotics Autom. Mag.*, vol. 17, no. 1, pp. 100–112, 2010.

[13] E. D. A. Silva, E. Valentin, J. R. H. Carvalho, and R. D. S. Barreto, A survey of model driven engineering in robotics, *J. Comput. Lang.*, vol. 62, pp. 101021, 2021.

[14] 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, *Inf. Technol.*, vol. 57, no. 2, pp. 85–98, 2015.

[15] N. Niknejad, W. Ismail, I. Ghani, B. Nazari, M. Bahari, and A. R. B. C. Hussin, Understanding service-oriented architecture (SOA): A systematic literature review and directions for further investigation, *Inf. Syst.*, vol. 91, pp.

101491, 2020.

[16] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, ROS: An opensource robot operating system, presented at ICRA Workshop on Open Source Software，Kobe，Japan，2009.

[17] S. García, D. Strüber, D. Brugali, T. Berger, and P. Pelliccione, Robotics software engineering: A perspective from the service robotics domain, in *Proc. 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, Virtual Event, USA, 2020, pp. 593–604.

[18] J. Xin, Y. Qu, F. Zhang, and R. Negenborn, Distributed model predictive contouring control for real-time multi-robot motion planning, *Complex System Modeling and Simulation*, vol. 2, no. 4, pp. 273–287, 2022.

[19] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. Boca Raton, FL, USA: CRC Press, 2018.

[20] M. Colledanchise, D. Almeida, and P. Ögren, Towards blended reactive planning and acting using behavior trees, in *Proc. 2019 Int. Conf. Robotics and Automation* (*ICRA*), Montreal, Canada, 2019, pp. 8839–8845.

[21] J. Kuckling, A. Ligot, D. Bozhinoski, and M. Birattari, Behavior trees as a control architecture in the automatic modular design of robot swarms, in *Proc. 11th Int. Conf. Swarm Intelligence*, Rome, Italy, 2018, pp. 30–43.

[22] B. G. Woolley and G. L. Peterson, Unified behavior framework for reactive robot control, *J. Intell. Robotic Syst.*, vol. 55, no. 2, pp. 155–176, 2009.

[23] S. Yang, X. Mao, Y. Lu, and Y. Xu, Towards a behavior tree-based robotic software architecture with adjoint observation schemes for robotic software development, *Autom. Softw. Eng.*, vol. 29, no. 1, pp. 31, 2022.

[24] D. Cassou, E. Balland, C. Consel, and J. Lawall, Leveraging software architectures to guide and verify the development of sense/compute/control applications, in *Proc. 33rd Int. Conf. Software Engineering*, Waikiki, HI, USA, 2011, pp. 431–440.

[25] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, The BRICS component model: A model-based development paradigm for complex robotics software systems, in *Proc. 28th Annual ACM Symp. Applied Computing*, Coimbra, Portugal, 2013, pp. 1758–1764.

[26] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge, UK: Cambridge University Press, 2016.

[27] M. Ghallab, D. Nau, and P. Traverso, The actor’s view of automated planning and acting: A position paper, *Artif. Intell.*, vol. 208, pp. 1–17, 2014.

[28] D. Kortenkamp, R. Simmons, and D. Brugali, Robotic systems architectures and programming, in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, eds. Cham, Switzerland: Springer, 2016, pp. 283–306.

[29] F. Michaud and M. Nicolescu, Behavior-based systems, in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, eds. Cham, Switzerland: Springer, 2016, pp. 307–328.

[30] P. Gregory, D. Long, M. Fox, and J. C. Beck, Planning modulo theories: Extending the planning paradigm, in *Proc. Twenty-Second Int. Conf. Autom. Plan. Sched.*, Atibaia, Brazil, 2012, pp. 65–73.

[31] S. Ross, J. Pineau, S. Paquet, and B. Chaib-Draa, Online planning algorithms for POMDPs, *J. Artif. Intell. Res.*, vol. 32, no. 2, pp. 663–704, 2008.

Shuo Yang received the PhD degree in software engineering from National University of Defense Technology, Changsha, China in 2022. He is currently a lecturer in control science and engineering at the College of Systems Engineering, National University of Defense Technology, Changsha, China. His current research interests include cyber-physical robotic system modeling, robotic software development, behavior tree modeling, and robotic task planning.

Qi Zhang received the PhD degree in control science and engineering from National University of Defense Technology, Changsha, China in 2018. He is currently a lecturer in control science and engineering at the College of Systems Engineering, National University of Defense Technology, Changsha, China. His research interests include behavior tree modeling and Computer Generated Forces (CGF) modeling and learning.