2026 IEEE/SICE International Symposium on System Integration (SII)
January 11-14, 2026. Cancún, México

CORAL: A Unifying Abstraction Layer for Compositional Robotics Software

Steven Swanbeck and Mitch Pryor

Abstract— **Despite the multitude of excellent software components and tools available in the robotics and broader software engineering communities, successful integration of software for robotic systems remains a time-consuming and challenging task for users of all knowledge and skill levels. And with robotics software often being built into tightly coupled, monolithic systems, even minor alterations to improve performance, adjust to changing task requirements, or deploy to new hardware can require significant engineering investment. To help solve this problem, this paper presents CORAL, an abstraction layer for building, deploying, and coordinating independent software components that maximizes compositionality to allow for rapid system integration without modifying low-level code. Rather than replacing existing tools, CORAL complements them by introducing a higher-level abstraction that constrains the integration process to semantically meaningful choices, reducing configuration complexity without limiting adaptability to diverse domains, systems, and tasks. We describe CORAL in detail and demonstrate its utility in integrating software for scenarios of increasing complexity, including LiDAR-based SLAM and multi-robot corrosion mitigation tasks. By emphasizing practical compositionality in robotics software, CORAL offers a scalable solution to a broad range of robotics system integration challenges, improving component reusability, system reconfigurability, and accessibility to both expert and non-expert users. We release CORAL open source[1].**

I. INTRODUCTION

A. Motivation

The difficulty of robotics system integration is widely appreciated within the robotics community, requiring interdisciplinary knowledge across computer science, mechanical and electrical engineering, and physics [1]. Even when considering only software, the design and integration of a robot is extremely complex, requiring careful coordination spanning low-level sensor and actuator drivers to high-level behavior orchestration [2]. Each of these components must be carefully engineered to function in concert to produce the desired system behavior, resulting in time-consuming and technically demanding system integration efforts, even for experienced roboticists. For non-experts, this presents a barrier that is often prohibitive. Due to this difficulty, ad hoc practices dominate in the robotics domain, increasing challenges to adapt or scale solutions across different systems, tasks, or applications [3].

To address these challenges, this paper introduces CORAL, a unifying abstraction layer that simplifies the development and deployment of robotics software by abstracting complex capabilities into composable modules that can be reliably deployed in new systems and applications. To do this, CORAL combines popular tools from the robotics and broader software engineering communities into a unified abstraction grounded in the mathematical principle of *compositionality* [4], which asserts that the behavior of a complex system can be fully determined by recursively reasoning about the structure of and interactions between its components. Each component in CORAL affords a set of *interfaces* that specify how external processes can interact with its internal functionality, enabling components to be engineered with deep domain knowledge while abstracting away implementation details from the end user.

This approach dramatically reduces the number of free variables exposed during system integration. Rather than requiring users to write and compile thousands of lines of imperative code, CORAL enables them to define high-level configurations using only the parameters that meaningfully influence system behavior. The design space remains effectively infinite due to the combinatorics of component composition, but it becomes more *functional* and *tractable* via targeted abstraction. In contrast to the complexity of traditional robotics software development–which, even for experienced roboticists, can devolve into a situation analogous to the infinite monkey theorem trying to find the right combination of parameters in high degree-of-freedom systems–CORAL ensures that system integration is a problem of reasoning about high-level interactions between subsystems rather than navigating low-level implementation details. This shift empowers end-users to become behavioral experts focused on *what* robots should do without getting caught up in the mechanics of *how* they do it, reducing the effort and knowledge required for successful integration of complex robotic systems.

B. Related Work

In many robotics applications, especially those where system performance is critical, engineers often neglect system quality attributes such as maintainability, interoperability, and reusability. As a result, reimplementing capabilities that are nearly identical to those previously developed in another system is a common task for robotics engineers. When original implementations are not designed for reuse, this leads to redundant work, frustration, and extended development cycles [5]. In academia, this situation is especially common, as the system engineering required to build flexible and interoperable software takes time and most projects are funded for short-term scientific results [5]. Further compounding this

The authors are with Texas Robotics and the Walker Department of Mechanical Engineering, The University of Texas at Austin, Austin, TX 78712, USA [stevenswanbeck, mpryor]@utexas.edu

[1]https://github.com/swanbeck/coral_cli.git

issue is the rapid evolution of robotics technologies, which discourages long-term investment in reusable infrastructure that may not be able to evolve to support new systems.

While the open-source community has developed an abundance of robotics software (nearly 900,000 repositories on GitHub are tagged with the "robotics" keyword), only a small subset meet the burden of code and documentation quality required to be broadly accessible and useful to the larger community [2]. Furthermore, even when excellent individual components exist, the lack of a widely used set of high-level paradigms or architectural conventions that consistently support composition means that integration into a larger system often requires substantial engineering investment. Bridging this "last mile" of system integration, particularly when synthesizing components designed for different original use cases, remains particularly challenging.

To make these problems more tractable, a well-established design pattern in engineering is decomposition–breaking down large problems into smaller, more interpretable components. This principle underlies component-based software design, which has become the de facto standard in robotics [6]–[10]. In this paradigm, software systems are built from mutually decoupled units that have well-defined interfaces and are each responsible for a specific function [11]. Effective reuse of such components depends on three factors: implementation quality, interface compatibility, and functional relevance [6]; a component may be of the highest quality, but it will not be reused if it does not provide a useful function, is difficult to understand, or cannot be easily integrated with other systems.

To manage integration challenges, the robotics community has adopted several standard tools and abstractions. Most pervasive among them is middleware, which facilitates interprocess communication. Robot Operating System (ROS) [12], [13] has emerged as the most popular middleware, with a reported adoption rate of 88.5% among service robotics practitioners [14]. Although alternative middleware approaches continue to be developed, including approaches to make middlewares more model-based and accessible to a wider range of practitioners [1], ROS remains the most widely used and supported [2].

Another tool that has become ubiquitous in modern software development is containerization, and tools such as Docker [15] and Podman have experienced adoption within the robotics community [16], [17]. [17] introduces a design flow based on ROS, Docker, and Kubernetes that clusters compatible applications with similar dependencies in containers to increase the scaling, reconfigurability, and availability of services and workloads. Their focus is on improving reliability in the integration of robotic systems while also addressing the high disk usage associated with maintaining multiple images with overlapping dependencies. In contrast, our approach does not target optimization of container design or clustering; rather, we propose a design paradigm centered on individually containerized, modular applications aimed at maximizing cross-system and cross-application functional reusability, along with an orchestration mechanism that supports flexible composition and deployment of these capabilities.

Similarly to component-based design, skill-based architectures have become a popular abstraction to organize and build robot functionality [18]. These approaches define a set of atomic, reusable *skills* that encapsulate perception, decision-making, planning, execution, and other fundamental capabilities, which help developers create more maintainable and reusable software [19]. Skills are closely related to *behaviors*, which are usually referenced in the context of *behavior trees* [20]. Behavior trees have gained broad appeal within the robotics community over the past decade because they enable the creation of structured and reactive control sequences while remaining highly readable and easy to interpret [21]. Additionally, past work has shown that behavior trees offer superior flexibility compared to traditional finite state machines (FSMs), with modification to behavior trees requiring fewer changes regardless of system complexity [22], [23]. Crucially, behavior trees are more compositional than traditional FSMs because new behaviors can be added without requiring global changes to the transition structure and logic [24].

C. Contributions

Based on our survey of these past works, our approach improves the compositionality of robotics software by proposing a new paradigm that uses Docker–one of the most popular tools in the software engineering community for dependency management and reliable cross-system deployment–, ROS–the most widely used robotics middleware–, and behavior trees–an increasingly popular abstraction for complex behavioral system design–to create CORAL, a simple and powerful abstraction layer that enables rapid integration of complex systems and easy reuse of functional components to reduce the difficulty of robotics system engineering for both experts and non-expert end-users. By encapsulating many of the thorny nuances of successful system integration and reducing the free variables to only those that meaningfully influence the behavior of the final system, CORAL enables users to reason about *what* robots should do rather than *how* they can do it, requiring no modification of compiled modules that run reliably across systems and are able to operate synergistically without *a priori* knowledge of each other before runtime. We describe CORAL in detail and provide two demonstrations of different scales and complexities, including a single-robot LiDAR-based SLAM task and a multi-agent corrosion mitigation task combining semantic SLAM, augmented reality (AR)-based user interaction, and 3D coverage planning, to demonstrate how CORAL simplifies integration of complex robotic systems by emphasizing compositionality.

II. CORAL

A. Overarching Design Choices

Based on the assertion that highly compositional systems are simpler to modify, reconfigure, and reuse than monolithic ones, CORAL unifies several existing tools that each support

composition at different levels to produce a top-level abstraction that maximizes overall robot compositionality.

Containerization: Similarly to microservice-oriented architectures, CORAL configurations are structured as compositions of small, independent components. As in many software engineering domains, each component is a containerized process, enabling reliable deployment across systems and reducing the complexity of dependency management by packaging each component with a runtime environment containing all its dependencies. Containerization within CORAL uses Docker [15]. These containerized components form the compositional units of CORAL at the system level, which are interfaced and coordinated together using the following abstractions.

ROS: As has become ubiquitous in research robotics and increasingly prevalent in industry, CORAL uses ROS [12], [13] to transfer information between its decoupled components. Using ROS, information can be transferred between processes crossing network and filesystem boundaries, which is critical to accommodate CORAL's heavy use of containerization and support scaling to distributed systems. Practitioners are recommended to use ROS2 [13] within CORAL configurations, though it is possible to integrate ROS1 [12] components into a CORAL configuration using a ROS Bridge component, as in Section III-B.

Behavior Trees: Behavior trees are increasingly popular within the robotics community due to their modularity, extensibility, and explainability [21], which enable expression of highly complex and reactive system behavior while remaining easily reconfigurable. These properties stem from the highly compositional nature of behavior trees [23], where the behavior of the overall system can be determined by recursively reasoning about the composition of its atomic behaviors under a set of fundamental control structures. Using CORAL, running components afford sets of behaviors that can be composed into arbitrarily complex behavior trees describing the flow of logical control that produces desired robot behavior. In this way, the behaviors afforded by running components form the compositional units of CORAL at the task level. Flow of information between behaviors is managed via coordinated reads and writes to a shared blackboard structure, as is the common practice using behavior trees. Practically, CORAL uses the popular BehaviorTree.CPP[2] library, which has become widely used in robotics applications [21], [25].

CORAL's use of these three tools means that, to integrate potentially extremely complex robotic systems using CORAL components, a user must generate as few as three high-level configuration files:

1) At least one file detailing the containerized processes used in the system, expressed in the Docker Compose YAML syntax with minor modifications for compatibility with the CORAL command-line interface (CLI) discussed below;
2) At least one file specifying runtime parameters required by ROS processes started within the containers, expressed in the ROS parameter YAML syntax;
3) At least one file describing the behavior tree(s) to be run expressed in the BehaviorTree.CPP XML syntax.

The parameters specified in these files are the only free variables exposed for system integration, producing a much more manageable design space that still has sufficient expressivity to reconfigure across domains, systems, and tasks. These files are also written in formats that are accessible to non-programmers and can be edited quickly without requiring any kind of code recompilation or similar.

B. Components

A running CORAL instance consists of a set of independent system components coordinated together using one or many overarching behavior trees. Each component is expected to be individually containerized and provide or utilize functionality within the larger system using interfaces that it exposes at runtime.

We classify these components into one of three distinct types:

1) `Executors` run user-engineered behavior trees utilizing functionality afforded by other components;
2) `Skillsets` afford sets of behaviors used within `Executors` and serve requests made by `Executors` using those behaviors;
3) `Drivers` are any components required to enable `Executors` or `Skillsets` that do not directly afford any behaviors to `Executors`.

These categories of system components are described in detail in the following sections.

1) `Executors`: Each `Executor` is designed to accept and execute a behavior tree built using behaviors afforded by the deployed `Skillsets`. The most common design pattern for our selected behavior tree execution library[2] involves compiling a priori knowledge of all the behaviors to be used during runtime into the execution program. This approach limits composability, as an execution program therefore cannot be created independently of its eventual use case. To overcome this challenge, each `Skillset` contains compiled shared libraries of behaviors and ROS interfaces that are extracted when it is first run. Then, these libraries are passed to and loaded by the `Executors`, enabling them to use all capabilities of the started `Skillsets` without knowledge of the `Skillsets` or their afforded capabilities prior to runtime. In this way, CORAL enables identically compiled `Executors` to run behavior trees built using *any* behaviors afforded by *any* running `Skillsets`.

2) `Skillsets`: Each `Skillset` affords a set of behaviors used by `Executors` to leverage its functionality. These behaviors are created by the `Skillset` designer and should abstract the core capabilities of the entire `Skillset` into a set of minimal, functional interfaces. The term *functional* is used here in alignment with the functional programming paradigm, which prefers creating small functions that do not mutate any internal state and can be composed in a modular fashion to produce much more complex programs.

[2]`https://github.com/BehaviorTree/BehaviorTree.CPP`

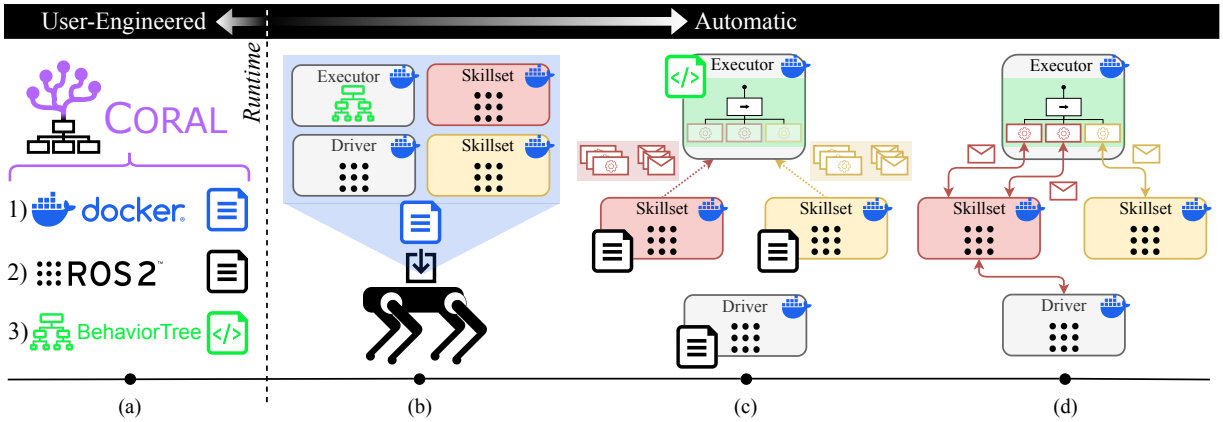

Fig. 1. **CORAL Runtime Process:** Diagrammatic representation of the CORAL runtime process for a simple system with a single `Executor`, two `Skillsets`, and one `Driver`. (a) Before runtime, a user engineers 1) a BehaviorTree.CPP XML-format behavior tree, 2) a Docker Compose/CORAL CLI YAML file, and 3) a ROS parameters YAML file. (b) Using the Compose YAML file, containers for each component are started on the deployment system. Any images not available on the deployment system are automatically pulled from a remote server, if available. (c) The user-engineered behavior tree is passed to the `Executor` and the ROS parameters are shared with the ROS nodes running inside the `Skillsets` and `Driver`. All `Skillsets` export compiled behavior and interface libraries that are loaded by the `Executor`. (d) Using the exported behaviors, the `Executor` constructs and runs the user-provided behavior tree, which uses the exported interfaces to interact with the started `Skillsets`.

In robotics, adherence to a strictly functional paradigm is impractical because there are many partially observed or unobserved states (the state of the physical world, for instance) that are difficult to accurately and completely capture in a program. However, an intentional effort to align behaviors with functional programming paradigms, where possible, increases the composability of these atomic elements in more complex systems. At runtime, afforded behaviors are extracted from each `Skillset` and made available to `Executors`. If afforded behaviors require any kind of processing or computation requiring dependencies beyond the C++ standard library or base ROS installation, they should call on ROS services or actions provided by running ROS nodes inside the affording `Skillset` rather than doing so internally. In this way, `Executors` only need to load compiled shared libraries for behaviors and any custom ROS interfaces used by the behaviors to leverage all the capabilities afforded by the running `Skillsets`. Additionally, each `Skillset` is recommended to be as hardware- and task-agnostic as possible to maximize its reusability. While this is infeasible for some robotics applications, which may be tightly coupled to a specific robot or task in a way that cannot be parameterized within the top-level configuration files, making an effort to abstract capabilities into operationally minimal atomic elements maximizes cross-application compositionality.

3) `Drivers`: `Drivers` describe any component that supports the `Executors` and/or `Skillsets` and does not execute or directly afford behaviors to be used within a behavior tree. Most commonly, `Drivers` include hardware-dependent modules that might, for example, stream sensor data from a camera or execute low-level joint motion in a manipulator. Rather than being functional behaviors, these capabilities are typically provided as input or output data streams. We refer to `Drivers` as *interchangeable* when they provide equivalent utility to the `Executors` and/or `Skillsets` in a CORAL configuration. For example, as shown in Fig. 2, a ROS2 bag player `Driver` providing a datastream of 3D point cloud data is interchangeable with the subcomposition of a ROS Bridge `Driver` and ROS1 Ouster LiDAR `Driver` connected to a live sensor.

C. Runtime

As previously described, prior to runtime a CORAL user must create at least one of each a BT.CPP XML syntax file, a Docker Compose and CORAL CLI-compatible YAML syntax file, and a ROS parameters YAML syntax file. If a user has properly constructed these files and has compiled and installed the CORAL CLI, an entire instance with all its `Executors`, `Skillsets`, and `Drivers` can be started with a single command. Because all components are containerized applications, they can be stored on a runtime-accessible server and automatically pulled if not found locally, allowing for easy deployment of software to a new system while avoiding the complexities of system-wide dependency management. At runtime, containers running the `Executor`, `Skillset`, and `Driver` processes are started using the engineered Docker Compose file(s). Each `Executor` receives a BehaviorTree.CPP tree and the ROS nodes in the `Skillsets` and `Drivers` are started using the parameters in the provided ROS parameters file(s). Immediately after all processes have been started, all dependencies required by the `Executor`(s) to interface with each `Skillset` are exported by that `Skillset`. These usually include compiled C++ libraries for the BehaviorTree.CPP behaviors afforded by the `Skillset` and any custom ROS interfaces the behaviors use to communicate with the processes running inside the affording `Skillset` container. After registering all dependencies, `Executors` can successfully run behavior trees composed of the behaviors afforded by all running `Skillsets`. This runtime process is visualized in Fig. 1.

Following the same paradigm of building behavior trees run by `Executors` using the capabilities afforded by `Skillsets` supported by `Drivers` with no dependence between components prior to runtime, this simple compositional paradigm can scale to complex robotic systems.

III. Demonstrations

CORAL's emphasis on maximizing compositionality means that it provides a solution to robot system integration problems that scales well with system complexity, providing modular reuse of atomic components, replacement of interchangeable components and subcompositions, and simple addition of existing configurations for new applications. We present two demonstrations to show these capabilities:

A. A LiDAR-based SLAM task in Section III-A that generates a point cloud map by playing a ROS bag file;
B. A multi-agent corrosion mitigation task in Section III-B that includes semantic SLAM, AR-based user interaction, and 3D coverage planning tasks performed by several coordinating robots.

Demonstration A was selected to show how CORAL's compositional properties enable a relatively simple composition performing LiDAR-based SLAM using a ROS Bag to be extended to the semantic SLAM task running onboard a robot in Section III-B by inserting new behaviors into the existing behavior tree, running new `Skillsets` and `Drivers`, and replacing the ROS Bag player with an interchangeable subcomposition consisting of ROS Bridge and ROS1 LiDAR `Drivers` with a live sensor. Demonstration B was selected to show how CORAL's emphasis on compositionality simplifies the integration of even distributed teams of robots by allowing previous compositions to be composed and supporting the reuse of atomic components between systems and tasks. Demonstration B shows how the semantic SLAM CORAL configuration is itself a composable unit that fits into a much larger and more complex multi-system composition, and how common components are shared between systems performing different tasks. The complete CORAL configurations and runnable examples for these demonstrations are available online[3].

A. LiDAR-based SLAM

The composition for this demonstration is shown in Fig. 2(a). It uses a single `Executor` running a behavior tree that loads existing map data, iterates a SLAM loop until a termination signal is received, and finally saves map data to disk. Two `Skillsets` are used, including a SLAM component based on [26] that supports map sharing over a network and dense local map extraction and a raw data aggregator component that reads sensor data from live sensors or bags and exposes a request-response service to obtain time-synchronized snapshots of this data. In Fig. 2, each behavior is colored to match its affording `Skillset`. A single `Driver` that plays ROS bag files completes the composition by providing a stream of point cloud data made available to the rest of the system via the raw data aggregator `Skillset`. The map generated in this demonstration using bag data from [27] is shown in Fig. 2(a). Readers can recreate this demonstration by running a single command[3].

[3]`https://github.com/swanbeck/coral_examples.git`

B. Multi-Agent Corrosion Mitigation

To demonstrate CORAL's ability to compose and deploy software for a more complex system, we consider the task of detecting and repairing corroded material in industrial environments using a heterogeneous team of robots. This application is inspired by our previous work [28] and is a real-world problem with tremendous financial and environmental impacts that robotic solutions can reduce. In [28], we identified that a heterogeneous team of systems with different capabilities would be best equipped to perform corrosion mitigation in industrial environments and faced challenges developing, deploying, and quickly adapting our software across robots and workflows. In this work, we consider a fixed team composition of the following systems:

- **Wheeled Survey Robot**: Navigates the environment and captures time-synchronized camera and LiDAR data. It processes image data through a corrosion detection model and then fuses the image and LiDAR snapshots to create point clouds with RGB and semantic labels. This data is incrementally integrated into a dense, labeled environment map, which is transmitted to the remote server upon completion of the survey.
- **Remote Server and AR Headset**: The server receives map data and mediates user interactions via the AR headset. These interactions allow for visualization of the generated map overlaid in physical space, placement of virtual volumes that mark regions of interest for downstream reconstruction, and confirmation or rejection of detections made by the survey robot. Once user interaction is complete, the downstream robots are authorized to complete their repair actions.
- **Quadruped and High-Reach Manipulator**: Upon receiving authorization, these systems pull map data from the server and begin to localize themselves, navigate to confirmed corrosion detections, and reconstruct the regions surrounding the detections or corresponding to markers placed by the human user. During reconstruction, additional images are captured and annotated using an onboard detection model. High-fidelity reconstructions are generated using depth images, from which virtual fixtures and optimized manipulation trajectories are generated. The robots can then execute these trajectories and simulate the application of a corrosion-inhibiting spray via a proxy laser device to complete the surface coverage task.

This workflow is summarized in Fig. 3.

The CORAL configuration for the semantic SLAM task performed by the survey robot is shown in Fig. 2(b). It builds on the LiDAR-based SLAM task in Fig. 2(a), adding new `Skillsets` for image processing using a custom corrosion detection model and fusion of image and point cloud data with corresponding additions to the SLAM loop within the

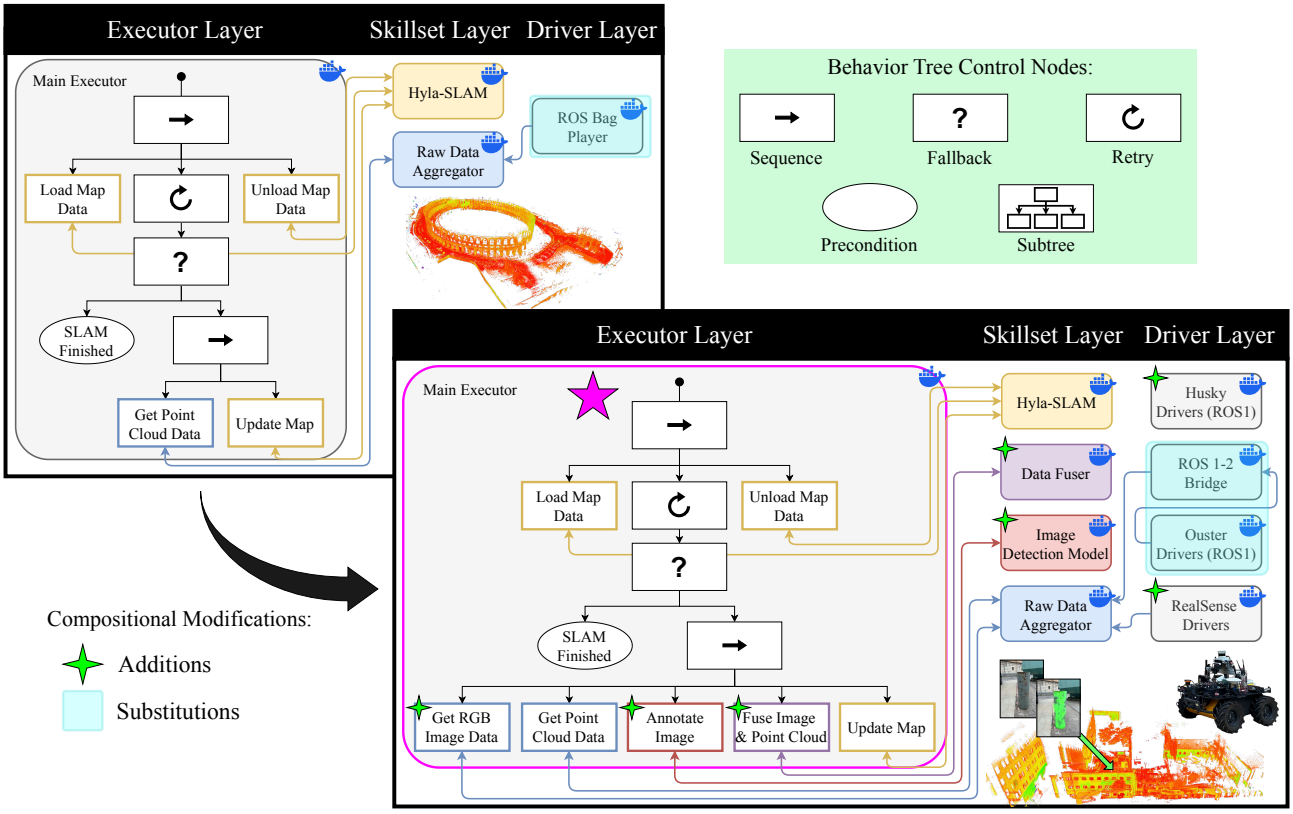

Fig. 2. **CORAL Compositions for LiDAR-Based SLAM:** (a) The composition for the LiDAR-based SLAM task in Section III-A that generates a map with [26] using ROS bag data from [27]. Behaviors made available at runtime share the color of the `Skillset` that affords them. The generated map is shown inset. (b) A revised composition for the semantic SLAM task in Section III-B. Compositional modifications from (a), including both additions of new behaviors, `Skillsets`, and `Drivers` to add semantic information and substitutions of `Drivers` to provide an equivalent LiDAR datastream are highlighted. The complete semantic SLAM behavior tree in (b) is also included as a subtree in Fig. 3 and highlighted.

behavior tree. Additionally, the ROS bag player used in Section III-A is replaced by LiDAR sensor and ROS bridge `Drivers` that together provide an equivalent stream of point cloud data from the live sensors onboard the robot. ROS parameters used by the `Skillset` nodes and the flow of information through the behavior tree are altered compared to Section III-A, but all required changes for this extension are made in the three high-level configuration files with no changes to low-level code or shared components.

The CORAL compositions for all systems in this demonstration are shown in Fig. 3. The behavior tree, `Skillsets`, and `Drivers` shown in Fig. 2(b) for semantic SLAM are integrated into the complete behavior tree and set of components used for the survey robot. Similar configurations are shown for the server with headset, quadruped, and high-reach manipulator systems. The overall system uses many `Skillsets` not described in detail, including a file server for coordinating over-network sharing of data on disk, an AR server that interfaces with an external AR application [29] to enable user interaction, a next-best-view server based on [30] that enables information gain-driven reconstruction of bounded regions of interest, a point cloud processing server that extracts the locations and geometries of predicted corrosion, and a virtual fixture generation server based on [31] that outputs sets of traversable virtual fixtures to perform surface coverage. Many additional `Drivers` are also used, including robot drivers for the Clearpath Husky, Boston Dynamics Spot, and hybrid Doosan H2017 systems, drivers for the camera and LiDAR sensors deployed on each system, a special middleware client [32] for communicating with the AR headset, and drivers running micro-ROS to communicate with Teensy microcontrollers used in custom laser end-effector attachments for coverage visualization.

Despite its complexity, all required software for this demonstration was integrated using generalizable CORAL components with just three files per robot totaling less than 1500 lines of high-level, uncompiled code. The team of robots completed its survey, user interaction, and surface coverage tasks using the engineered system composition, as shown in Fig. 3. In comparison to previous integration efforts using a single robot to perform corrosion mitigation with a monolithic, densely integrated software stack [28], CORAL's composition was much faster to iterate during testing, quickly reconfigure for updated task requirements, and deploy to new systems. Readers are referred to the complete CORAL configuration[3] for additional details.

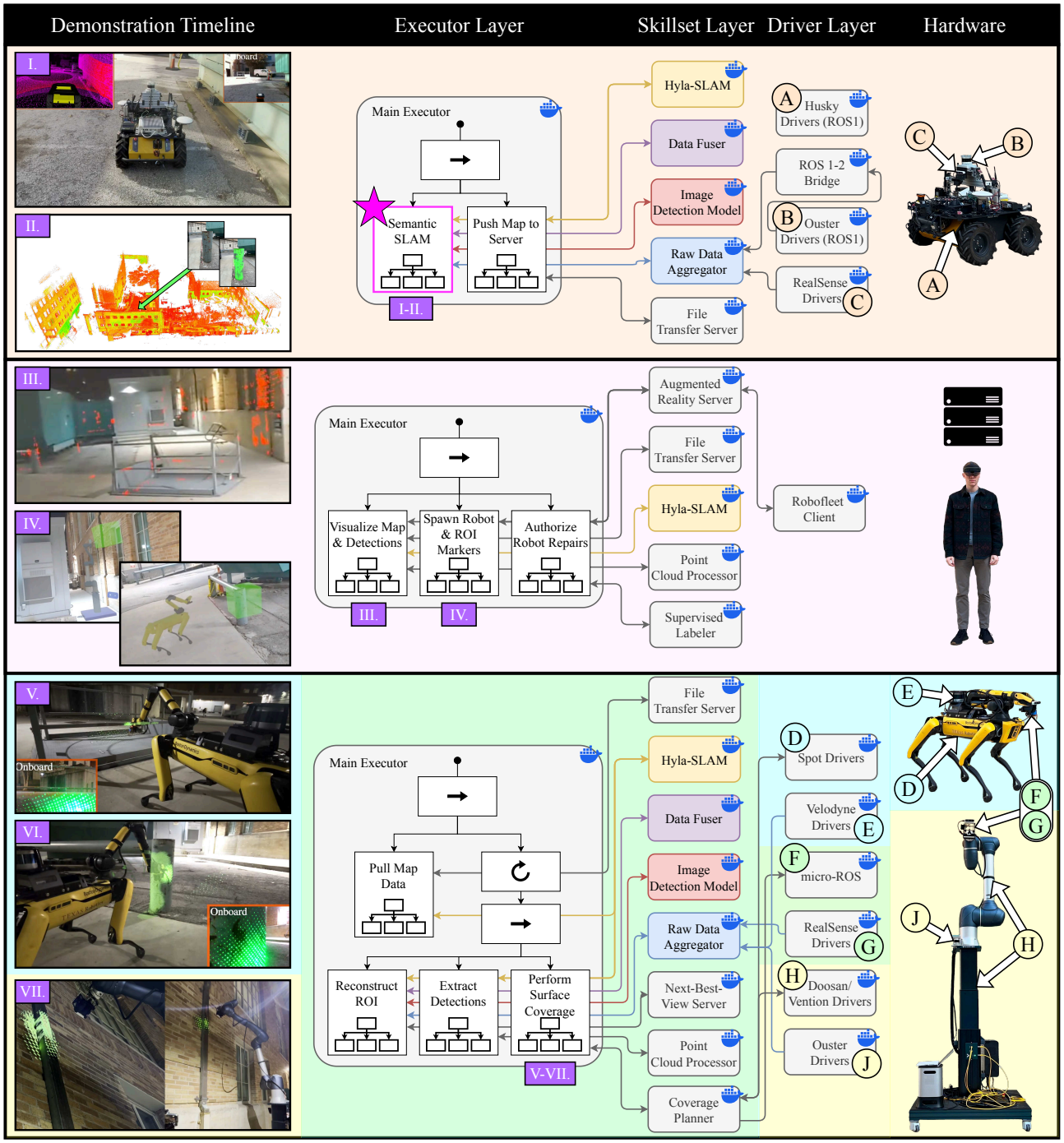

Fig. 3. **CORAL Composition for Multi-Agent Corrosion Mitigation:** The multi-agent corrosion mitigation demonstration in Section III-B unfolds as shown in panels I)–VII). I) A wheeled robot is used for initial surveying and generates a 3D map containing corrosion detections using the subcomposition from Fig. 2(b). The behavior tree from Fig. 2(b) is show as a subtree and highlighted in the survey robot's configuration and its `Skillsets` and `Drivers` are a superset of those in Fig. 2(b). II) Once the map is generated, it is pushed to a remote server, enabling III) visualization of detections overlaid on the physical environment using an augmented reality headset. IV) Using the headset, a human supervisor can confirm detections, spawn virtual robot models to assess base placement and reachability, and spawn region-of-interest (ROI) volumes used for downstream repair tasks. V-VII) Once the supervisor is finished, control flows to the quadruped and high-reach manipulator systems, which pull the map data from the server, begin localizing and navigate or are moved to positive corrosion detections, accurately reconstruct the contents of the placed ROI volumes using an information gain-driven procedure, and generate and execute plans for surface coverage over detected corrosion. `Skillsets` from Fig. 2 retain their colors while new `Skillsets` are left uncolored. Labeled subtrees are used to show the general flow of control throughout the demonstration. Readers are referred to the complete configuration[3] for full behavior trees and implementation specifics.

IV. Conclusions

This paper presents CORAL, an abstraction layer for robotics software that unifies already popular tools in the robotics and software engineering communities to simplify integration of complex robotic systems by maximizing compositionality across the system and task levels. By abstracting robotics software into `Executor`, `Skillset`, and `Driver` components that follow a set of standardized abstractions, we show how highly specialized systems can be composed from independent modules using just a few high-level configuration files. By exporting compiled dependencies at runtime, CORAL enables software to run synergistically without requiring prior knowledge of available components. Using `Skillsets` that expose a set of minimal, functional behaviors, users can design elaborate behavior trees that leverage capabilities afforded during runtime to produce complex and reactive system behavior. All components are also containerized, allowing for both reliable execution on new systems and simple deployment via a runtime-accessible server. We show how these compositional abstractions scale to complex and specialized applications in LiDAR-based SLAM and multi-agent corrosion mitigation tasks and provide code and examples open source.

Acknowledgment

We thank Jorge A. Diaz, Fabián Parra Gil, Alex Navarro, and Mathew Huang for sharing source code that was packaged into CORAL `Skillsets` and `Drivers` for the multi-agent corrosion mitigation demonstration in this paper.

References

[1] K. Panayiotou, E. Tsardoulias, C. Zolotas, A. L. Symeonidis, and L. Petrou, "A framework for rapid robotic application development for citizen developers," *Software*, vol. 1, no. 1, 2022.

[2] J. Haviland and P. Corke, "Robotics software: Past, present, and future," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 7, 2024.

[3] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Pelliccione, and T. Berger, "Software variability in service robotics," *Empirical Softw. Eng.*, vol. 28, no. 2, 2023.

[4] A. Censi, J. Lorand, and G. Zardini, *Applied Compositional Thinking for Engineers*. 2022, Work in Progress Book.

[5] D. Brugali and E. Prassler, "Software engineering for robotics," *IEEE Robot. Automat. Mag.*, vol. 16, no. 1, 2009.

[6] D. Brugali and P. Scandurra, "Component-based robotic engineering (part I)," *IEEE Robot. Automat. Mag.*, vol. 16, no. 4, 2009.

[7] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part II)," *IEEE Robot. Automat. Mag.*, vol. 17, no. 1, 2010.

[8] O. Rogovchenko and J. Malenfant, "Composition and compositionality in a component model for autonomous robots," in *Proc. Int. Conf. Softw. Composition*, 2010.

[9] C. Schlegel, A. Lotz, M. Lutz, and D. Stampfer, "Composition, separation of roles and model-driven approaches as enabler of a robotics software ecosystem," *Software Engineering for Robotics*, 2021.

[10] F. Gosselin, G. Acher, B. Gradoussoff, *et al.*, "An intelligent robotics modular architecture for easy adaptation to novel tasks and applications," in *Proc. IEEE Int. Conf. Automat. Sci. Eng.*, 2023.

[11] D. Calisi, A. Censi, L. Iocchi, D. Nardi, *et al.*, "Design choices for modular and flexible robotic software development: The OpenRDK viewpoint," *J. Softw. Eng. Robot.*, vol. 3, no. 1, 2012.

[12] M. Quigley, B. Gerkey, K. Conley, *et al.*, "ROS: An open-source robot operating system," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2009.

[13] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," *Sci. Robot.*, vol. 7, no. 66, 2022.

[14] S. García, D. Strüber, D. Brugali, T. Berger, and P. Pelliccione, "Robotics software engineering: A perspective from the service robotics domain," in *Proc. ACM Joint Meeting on European Softw. Eng. Conf. and Symp. Found. Softw. Eng.*, 2020.

[15] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 239, no. 2, 2014.

[16] F. Lumpp, M. Panato, F. Fummi, and N. Bombieri, "A container-based design methodology for robotic applications on Kubernetes edge-cloud architectures," in *IEEE Forum on Specification & Design Languages*, 2021.

[17] F. Lumpp, M. Panato, N. Bombieri, and F. Fummi, "A design flow based on Docker and Kubernetes for ROS-based robotic software applications," *ACM Trans. Embedded Comput. Syst.*, vol. 23, no. 5, 2024.

[18] M. Mayr, F. Rovida, and V. Krueger, "SkiROS2: A skill-based robot control platform for ROS," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2023.

[19] A. Albore, D. Doose, C. Grand, J. Guiochet, C. Lesire, and A. Manecy, "Skill-based design of dependable robotic architectures," *Robot. Auton. Syst.*, vol. 160, 2023.

[20] K. Winter, I. J. Hayes, and R. Colvin, "Integrating requirements: The behavior tree philosophy," in *Proc. IEEE Int. Conf. Softw. Eng. Formal Methods*, 2010.

[21] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and AI," *Robot. Auton. Syst.*, vol. 154, 2022.

[22] M. Iovino, J. Förster, P. Falco, J. J. Chung, R. Siegwart, and C. Smith, "On the programming effort required to generate behavior trees and finite state machines for robotic applications," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2023.

[23] M. Colledanchise and P. Ögren, "How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees," *IEEE Trans. Robot.*, vol. 33, no. 2, 2016.

[24] M. Iovino, J. Förster, P. Falco, J. J. Chung, R. Siegwart, and C. Smith, "Comparison between behavior trees and finite state machines," *IEEE Trans. Automat. Sci. Eng.*, 2025.

[25] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The Marathon 2: A navigation system," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2020.

[26] S. Swanbeck and M. Pryor, "Hyla-SLAM: Toward maximally scalable 3D LiDAR-based SLAM using dynamic memory management and behavior trees," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2025.

[27] L. Brizi, E. Giacomini, L. D. Giammarino, *et al.*, "VBR: A vision benchmark in Rome," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2024.

[28] S. Swanbeck, J. Rosenbaum, J. A. Diaz, F. Parra Gil, and M. Pryor, "Fighting rust with robots: Toward scalable corrosion mitigation in industrial environments using robotic systems," in *Proc. IEEE Int. Symp. Saf. Secur. Rescue Robot.*, 2025.

[29] F. Regal, C. Petlowany, C. Pehlivanturk, *et al.*, "AugRE: Augmented robot environment to facilitate human-robot teaming and communication," in *Proc. IEEE Int. Conf. Robot Human Interactive Communication*, 2022.

[30] A. Bircher, M. Kamel, K. Alexis, H. Oleynikova, and R. Siegwart, "Receding horizon "Next-Best-View" planner for 3D exploration," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2016.

[31] A. Sharp and M. Pryor, "Virtual fixture generation for task planning with complex geometries," *ASME J. Comput. Inf. Sci. Eng.*, vol. 21, no. 6, 2021.

[32] K. S. Sikand, L. Zartman, S. Rabiee, and J. Biswas, "Robofleet: Open source communication and management for fleets of autonomous robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2021.