Carnegie Mellon University

Thesis Proposal:
Automatic Inference of Behavioral Component Models for ROS-Based Robotics Systems

Tobias Dürschmid
Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA

Thesis Committee:
Claire Le Goues, Co-Chair
David Garlan, Co-Chair
Christopher Timperley
Ivano Malavolta

Submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy in Software Engineering

May 2, 2024

Abstract

Robotics systems are complex component-based systems, which can consist of many interacting components. When composing and evolving complex component-based systems, the resulting behavior of component interactions sometimes differs from the developers' expectations. This can, for example, manifest itself in components indefinitely waiting for a required message that no other component sends, components reaching a deadlock state, or messages getting ignored due to systems being in an incorrect state. These bugs, which we call *behavioral architecture composition bugs*, are often hard to find, because the fault locations are spread throughout many different locations in the system.

Model-based analysis is a common technique to identify incorrect behavioral composition of complex, safety-critical systems, such as robotics systems. However, in practice, robotics companies usually do not have any formal models, as models for hundreds of software components is a very costly and often labor-intensive and error-prone process. Behavioral models, which would need to be updated whenever the behavior of the component changes, are especially expensive to create.

In this thesis proposal, I present an approach to automatically infer behavioral models for components of systems based on the Robot Operating System (ROS), the most popular framework for robotics systems, using a combination of static and dynamic analysis by exploiting assumptions about the usage of the ROS framework Application Programming Interface (API) and behavioral idioms. Static analysis looks for architecturally-relevant API calls that implement message sending, handing received messages, sleeping for a periodic interval, and behavioral idioms that implement state-dependent behavior and state transitions. Based on this information, static analysis infers state machine models of architecturally-relevant component behavior. Due to limitations of static analysis, the resulting models are often partial. To complete statically inferred models, I propose to instrument the source locations of known unknowns and dynamically observe their values. Then, resulting models will be translated into the common language TLA+/PlusCal used for model-checking. Furthermore, I propose a model-based analysis technique to find architecture-misconfiguration bugs in the resulting TLA+ models. Finally, I propose an end-to-end evaluation that measures the effectiveness and practicality of these contributions.

This work is a contribution towards making well-proven and powerful but infrequently used methods of model-based analysis more accessible and economical in practice to make robotics systems more reliable and safe.

Contents

Part I

Context

1 Introduction

Ensuring that robotics systems operate safely and correctly is an important challenge in software engineering. As robots are becoming increasingly integrated in work environments and the daily lives of many people [43, 59, 85, 36] their faults can potentially cause dramatic harm to people [8, 41, 84]. However, ensuring that robotics systems are safe and operate correctly is hindered by their large size and complexity [50, 62, 1, 2]. Many robotics systems are comprised of hundreds of thousands of lines or millions of lines of code [82].

Robotics systems, especially systems written for the Robot Operating System (ROS) [73], the most popular robotics framework, are often *component-based*, i.e., are implemented as independently deployable run-time units that communicate with each other primarily via messages [3, 44, 19, 73, 82]. They can be comprised of hundreds of software components, each of which can have complex behavior [14, 52, 82]. Many ROS systems are predominantly composed of reusable component implementations created by external developers [49]. In this context, the main challenge is their correct composition [82, 20].

The composition and evolution of software components is error-prone, since components regularly make undocumented assumptions about their environment, such as receiving a set of initialization messages before starting operation. When composed inconsistently, the behavior of these systems can be unexpected, such as a component indefinitely waiting, not changing to the desired state, ignoring inputs, message loss, or publishing messages at an unexpectedly high frequency [34, 20, 83]. In this thesis proposal we call these bugs "*behavioral architectural composition bugs*", because they are caused by inconsistent compositions and impact the software architecture's behavior. Finding and debugging behavioral architectural composition bugs in robotics systems is usually challenging, because components frequently fail silently and failures can propagate through the system [50, 38, 1, 2].

Software architects commonly use model-based architecture analysis to ensure the safety and correct composition of components [25, 18, 67, 68, 86, 58, 7, 39]. Model-based analysis is a design-time technique to evaluate whether design options meet desired properties. Systems are modeled as a set of interconnected views, such as *component-connector models* (describing whats components are in the system, what ports they have, and how ports are connected between components), *behavioral views* (state machines or activity diagrams describing the dynamic reaction a component can have to receiving messages at its ports), and *deployment views* (mapping component instances to processing units) [22]. Using models of interconnected views of the current system, architects can find inconsistencies or predict the impact of changes on the system's behavior.

However, in practice, due to the complexity of robotics systems, creating models manually is time-consuming and difficult [86, 24, 25]. This motivates work on automated model recovery to reduce the modeling effort and make formal analysis more accessible in practice.

Architectural recovery techniques, such as `ROSDiscover` [82], HAROS [76, 74], and the tool by Witte et al. [88], can reconstruct component-connector models. Such reconstructions can be effective in finding some bugs resulting from topic name misspellings or connectors that connect ports of different message type. However, they do not reconstruct behavioral models. Without behavioral models model-based analysis cannot reason about dynamic aspects that describe how the components react to inputs and how

they produce outputs, such as whether a component sends a message in response to receiving an input, whether it sends messages periodically or sporadically, and what state conditions or inputs determine whether it sends a message.

Existing approaches for inferring behavioral models, such as Perfume [69], use dynamic analysis to infer state machines from execution traces. However, these approaches cannot guarantee that the relationships they find are causal since they observe only correlations within behavior.

To address the challenge of automatically inferring behavioral component models for ROS-based systems, we propose to use a combination of static analysis of the system's source code written in C++ and dynamic analysis of systematically generated execution scenarios based on results of static analysis.

In general, inferring behavioral models is undecidable [56]. Even a partial solution is practically challenging, because the analysis needs to infer what subset of arbitrary C++ code gets compiled to be executed as a single component, what subset of this component's code communicates with other components, and under what situations this code for inter-component-communication is reachable.

Fortunately, the following observations about the ROS ecosystem make this problem tractable for most cases in practice:

1. Component architectures and behaviors are defined via Application Programming Interface (API) calls that have well-understood architectural semantics [75].

2. The composition and configuration of components to build larger systems is done in separate architecture configuration files (i.e., launch files). Most of these result in "quasi-static" systems. That is, architectures rarely change following run-time initialization [75].

3. Behavioral patterns, such as periodically sending messages, are usually implemented using features provided by the ROS framework. Hence, most instances of those patterns follow a similar implementation pattern.

1.1 Thesis Statement

My thesis statement is:

Thesis Statement

"Assumptions about framework-specific APIs and idioms enable the automatic inference of practically useful behavioral component models for ROS-based robotics systems."

The terms used in this statement have the following definitions:

Assumptions about framework-specific APIs and idioms represent observations made about how developers in the ROS ecosystem implement architecturally-relevant behavior that are true for most cases but not necessarily all cases.

Automatic inference denotes a combination of static and dynamic analysis that requires only minimal human intervention and relies on the availability of source code, build artifacts, and a simulated execution environment.

Practically useful is characterized by the effectiveness to which the proposed approach can assist ROS developers during model-based analysis tasks that are common in their domain.

Behavioral component models are formal models describing architecturally-relevant behavior of ROS components as input-output state machines of their ports.

The thesis statement will be considered proven after I have developed a technique that infers behavioral component models based on assumptions about framework-specific APIs and idioms and evaluated its practical usefulness via an end-to-end evaluation on real-world ROS bugs that require behavioral component models to be found.

1.2 Contributions

More concretely, to provide evidence for the thesis statement I propose the following main contributions:

1. A **static analysis approach** that infers component behavior models from ROS code by looking for API calls of architecturally-relevant behavior and common idioms to implement common behavioral patterns in the ROS ecosystem. This contribution has been completed already and is presented in Chapter 5. It presents evidence towards the thesis statement by showing that assumptions about framework-specific APIs enable the automatic inference of partial behavioral component models for ROS-based robotics systems.

2. A **dynamic analysis approach** that enhances statically inferred models by designing systematic experiments to observe the behaviors of incomplete models. More details on how this proposed work might be implemented is described in Chapter 6. This contribution will provide further evidence towards the thesis statement by demonstrating that framework-specific assumptions can be used to complete partial behavioral component biased via dynamic analysis.

3. A **model-based analysis** of the resulting models that translates them into executable PlusCal/TLA+ models with a set of commonly requested analyses from the robotics domain. This proposed work is described in Chapter 7. This contribution will provide initial evidence of the usefulness of the inferred models by demonstrating how they can be used to automatically find bugs.

4. An **end-to-end evaluation** of the presented approach measuring the effectiveness of finding bugs and the practical usefulness including a **novel data set** of real-world architecture misconfiguration bugs in open source ROS systems. The completed work on the data set collection is described in Chapter 4. The proposed work of the end-evaluation is described in Chapter 8. This contribution will provide the final piece of evidence to prove the thesis statement by providing an empirical validation of the practical usefulness of inferred models.

1.3 Structure of Thesis Proposal

The remainder of the thesis proposal is structured as follows:

- Chapter 2 explains the required **background** on modeling and analysis of software architectures, robotics, and ROS that is required to deeply understand the contribution of this thesis proposal and the most closely **related work**.
- Chapter 3 gives a high-level **overview of the presented approach** to infer component-behavioral models for ROS systems by making assumptions on the framework and ecosystem.
- Chapter 4 presents the **bug data set** of architecture misconfiguration bugs used for evaluation and to demonstrate the existence and variety of architecture misconfiguration bugs.
- Chapter 5 describes our **static analysis approach** to infer component behavior models for ROS systems and evaluates the accuracy of this approach to support the thesis statement for static analysis.
- Chapter 6 outlines proposed work on using **dynamic analysis** complete the statically inferred models by exploiting assumptions about the ROS framework.
- Chapter 7 proposes work to demonstrate the usefulness of the inferred models by describing domain-specific **model-based analyses** that analyze properties of interest to roboticists on PlusCal/TLA+ as well as a translation from our inferred models to PlusCal/TLA+.

2 Background and Related Work

This chapter describes the relevant background in software architecture, model-based analysis, and robotics on which the proposed contributions rely, existing work on which we build on, and related work that solves similar problems in different ways.

2.1 Views in Software Architecture

This section explains the background on architectural views that is needed to understand the differences between module views, component-connector views, dynamic views, and deployment views.

Software architectures are usually represented as a set of views that describe different high-level aspects of the system.

2.1.1 Module View

The *module view*, also known as code view, displays the software the way programmers interact with the source code. It contains source code elements, such as packages, classes, methods, or data entries and their relationships, such as "is part of", "uses", or "contains". Hence, it shows the composition of a software into structural implementation units and can be visualized as Unified Modeling Language (UML) class diagrams, package diagrams, or other notations. It is often used to reason about concerns such as modularity, testability, or the location of a piece of code in the context of the high-level architecture.

To correctly reconstruct a module view an automated architectural recovery approach only needs to look at the source code of a system.

2.1.2 Component-Connector View

The *component-connector view*, also known as run-time architecture, represents a structural configuration of the architecture at run time.

It contains execution units known as *components*, such as processes, objects, or data stores and their interaction channels known as *connectors*, such as pipes, publish-subscribe, or call-return.

Hence, it shows a configuration of the architecture at run time. It is often used to reason about quality attributes such as performance, availability, or the correctness of a system configuration.

To correctly reconstruct a component-connector view of a system an architectural recovery technique usually needs to look at the executables generated by the compiler or needs to make specific assumptions on the development framework and architectural styles used to generate the software.

In this thesis proposal, the term *component* refers to executable run-time elements, not code modules.

2.1.3 Behavioral View

The *behavioral view*, also known as dynamic view, expresses the behavior of the system or its parts. It can describe input-output relationships for components, states, state transitions, actions. Hence, it shows

how the components react to inputs. It is often used to reason about concerns such as deadlock freedom, liveness properties, and safety properties.

To correctly reconstruct a behavioral view an automated architectural recovery approach needs to reason about run-time properties of the system.

2.2 Architectural Styles

This section describes common architectural styles in a non-ROS-specific way.

2.2.1 Publish-Subscribe

Publish-subscribe is an architectural style for asynchronous message sending that loosely couples senders (i.e., *publishers*) from receivers (i.e., *subscribers*) via a known intermediary interface (i.e., *publish-subscribe connector*) that functions as a layer of indirection.

After subscribing to a message type, a connector is added between the subscriber and all publishers of the corresponding message type and the subscriber starts to receive the messages whenever any corresponding publisher sends them. Depending on the implementation or configuration, subscribers also receive all messages previously published at the message type. Unsubscribing removes the connector to the corresponding subscriber.

Publish-subscribe is intended to allow for dynamic reconfiguration of the architecture during run time or reduction of syntactic dependencies between component implementations to increase reusability, changeability, and extensibility. However, due to the late binding of the connector at run time, the use of publish-subscribe for use cases that do not require reconfiguration during run time can suffer from architecture misconfiguration bugs. Publish-subscribe can also increase the latency of message delivery, reduce the certainty of delivery times when scaling vertically, and result in race conditions due to uncertain message ordering.

2.3 Robotics Systems

Robotics systems are complex, component-based cyber-physical systems. They often involve processing sensor data (perception), periodically analyzing the current situation in which the robot is to create actions (planning), and translating actions into actuator commands (control). Due to the nature of the domain, having multiple components that process and produce data of the same type, they often predominantly use publish-subscribe connectors for the benefits on flexibility and loose coupling. The flexible architecture, complexity of the domain, and complexity of the implementation often results in hard-to find bugs.

Static analysis and formal model-based analysis have been used to automatically find bugs in robot systems before [4, 58, 7, 39, 71]. For example, the systems Phriky [70], Phys [46], and Physframe [45] use type checking to find inconsistencies in assignments based on physical units or 3D transformations in ROS code.

Furthermore, Swarmbug [42] finds configuration bugs in robot systems that result from misconfigured algorithmic parameters, causing the system to behave unexpectedly.

These approaches focus on the analysis of bugs that result from coding errors that are localized in a

few places of the system. In contrast, our work aims to reconstruct models that can be used to identify incorrect composition or connection of components and therefore focus on architectural bugs.

2.4 The Robot Operating System (ROS)

ROS is the most popular open-source framework for component-based robotics systems. ROS has been deployed to a diversity of robots, including autonomous vehicles, mobile manipulators, underwater systems, and humanoids[1] in environments ranging from industrial warehouses to the International Space Station [60, 9].

To increase the reusability of the more than 7 400[2] software packages in the ROS ecosystem, ROS uses configuration mechanisms and connectors that are not bound during compile time but rather during run time [27]. These mechanisms include string-based identifiers for topics, services, actions, and parameters, as well as remappings between these identifiers [75].

A reference of the architecturally-relevant API calls is described in Appendix A.

2.4.1 Components in ROS

Components in ROS are called *nodes*. They can be defined and named via there `ros::init` API call. In ROS 1 each node runs in its own process. A special kind of node, called a *nodelet*, are nodes that run within the process of a parent node.

2.4.2 Connectors in ROS

ROS implements variants of the architectural styles presented in Section 2.2.

Topics Implement a Publish-Subscribe Style

Topics implement a publish-subscribe style providing asynchronous message-based, multi-endpoint communication between nodes. Nodes subscribe to topics using the string-representation of their name and name space. Then they receive any data published to the subscribed topics. There can be multiple publishers and subscribers for a topic. Topics are the main form of communication between nodes in ROS, and are used for periodic information (e.g., sensor data or positions) or sporadic requests, such as turning off a motor.

Services Implement a Synchronous Call-Return Style

Services implement a synchronous call-return style of communication between nodes. Nodes attempting to call a service look up the service provider in a registry based on the string-based name of the service. Due to the synchronous blocking behavior, services are intended for short queries, such as the state of a node or short mathematical computation.

1 https://robots.ros.org [Date Accessed: 18th August 2021]
2 https://index.ros.org/stats/ [Date Accessed: 3th May 2023]

Actions Implement an Asynchronous Call-Return Style

Actions implement an asynchronous call-return style for long-lived requests to be performed by another node. Nodes submit goals to other nodes (such as navigating to a particular location), and can register callbacks to keep apprised of feedback, and results. In ROS 1, actions are implemented as a library that uses the other two communication mechanisms.

2.5 Model-based Analyses

Software architects commonly use model-based architecture analysis to ensure the safety and correct composition of software components [3, 18, 67, 68, 86, 58, 7, 39]. Model-based analysis is a design-time technique to evaluate whether designs meet desired properties. Systems are modeled as a set of interconnected views, such as behavioral views (e.g., state machines or activity diagrams), and component-connector views, and deployment views [22]. Based on models of the current architecture of the system, software architects can find architectural inconsistencies or model changes to predict their impact on the system's behavior.

There has been a large amount of work on model-based analysis of software architectures based on component-connector models [51, 10, 11, 13, 81, 15, 48] and state machines [54, 33, 30, 5]. Since the models we infer follow the same format, our approach makes analyses like these more accessible to developers by reducing the effort to create the models.

2.6 Inference of Module Views

Most approaches for static recovery of software architectures reconstruct structural views of software modules from the perspective of a developer [78, 72, 12, 37, 26, 61, 64, 63, 6, 23, 29, 21]. The results from these approaches can be used to show architects the relative location of a piece of code in the module view of the architecture and ensure the consistency of dependencies [35, 80, 31]. Since module views present the the code before compilation, they cannot show the relationships of components during run time [22].

2.7 Inference of Component-Connector Models

`ROSDiscover` [82], HAROS [76, 74], and the tool by Witte et al. [88] can reconstruct component-connector models for robotics systems. Component-connector models describe the types of inputs that a component receives, the types of outputs it produces, and to what other components their input and output ports are connected to. However, component-connector models do not contain information about how a component reacts to inputs (e.g., what kind of output it produces in response to an input), whether an output port is triggered sporadically or periodically, and whether the component's behavior is dependent on states. Therefore, component-connector models cannot be used to analyze the data flow within a system.

2.8 Inference of Behavioral Models

Behavioral models of components can be inferred using dynamic analysis by observing the component behavior of representative execution traces. For example, DiscoTect [77] and Perfume [69] construct state machines from event traces. Similar approaches also use method invariants [53], Linear Temporal Logic (LTL) property templates [57] to increase the effectiveness. Domain-specific approaches have been proposed for for CORBA systems [66] or telecommunication systems [65]. The main limitation of dynamic approaches that are based on observation alone is that they can measure only correlations between inputs and states and outputs and cannot make claims about causal relationships. Additionally, approaches relying on dynamic execution might miss cases in rarely executed software. In contrast to this, our approach analyzes the control and data flow of the source code and therefore has the capabilities to differentiate concurrent behavior that just coincidentally happens after an input or state change from behavior that is control-dependent.

Furthermore, existing dynamic approaches need to execute a large number of representative traces through the system in real time, which can increase the time and cost of the model creation for computation-intense systems. In contrast, the approach I am proposing only executes the parts of the program that cannot be recovered using static analysis. This reduces the number of traces that need to be executed there minimizes the overall analysis time.

3 Overview of the Approach

3.1 Architecturally-relevant Behavior

Figure 3.1 (a) shows an example of a bug from the Autoware.AI [47] system in which the lattice_trajectory_gen component requires an input to perform its main functionality although no other component sends this message. Hence, lattice_trajectory_gen waits indefinitely.

Existing approaches that recover only component-connector models, such as `ROSDiscover` [82], cannot find this bug, because they cannot infer that the input is *required*, i.e., component's main functionality depends on it.

Fortunately, only a small part of the overall behavior of a component is relevant to describe the component's behavior on an architectural level. This makes it practically possible for static analysis to infer behavioral component models for complex systems.

To define the semantics of the behavioral models that `ROSInfer` infers, this section introduces the formalism of behavioral component models that I will use throughout the proposal.

Architecturally-Relevant Component Behavior

Architecturally-relevant component behavior is the set of all behaviors required to describe what causes a component to send messages (e.g., triggers, state variables, state transitions).

Component State Machine $C \coloneqq (S, s_0, I, O, \delta)$

A component state machine C is a 5-tuple of states S, an initial state $s_0 \in S$, input triggers I, outputs O, and transitions δ.

Component States $S \coloneqq [var_1 : B_1, var_2 : B_2, ..., var_n : B_n]$

Component states S are records of named state variables $var_1, var_2, ..., var_n \in String$ with types $B_1, B_2, ..., B_n \in Types$.
$Types \coloneqq \{Bool, Int, Enum, Float, String\}$.

Input Triggers $I \subseteq M_{in} \cup P \cup E$

An input trigger is a message handled by an **input port $m \in M_{in}$**, a **periodic trigger $p_f \in P$** with frequency $f \in Float$, or a **component event $e \in E$**, such as "component started". To keep the model simple, the content of messages is not modeled.

Outputs $O \subseteq M_{out} \cup \{\epsilon\}$

Outputs are either messages sent through output port $m \in M_{out}$ or the empty output ϵ for transitions that change only the state but do not produce an output.

Transition Function $\delta \coloneqq S \times I \rightharpoonup O \times S$

The partial transition function $\delta \coloneqq S \times I \rightharpoonup O \times S$ is represented in pre- and post-condition form with preconditions being predicates on $s \in S$ and $i \in I$ that define for which inputs and states the transition is triggered and post-conditions defining an output $o \in O$ and the next state $s' \in S$ in terms of s and i.

Unknown Value $\top$

Finally, the formalism needs a special element $\top$ (pronounced "top") that is used to represent an unknown value for cases in which the static analysis is unable to infer the value of an expression (e.g., the frequency of periodic publishing, values of initial states, or the right side of assignments of state variables). It is included in all data types: $\forall T \in Types : \top \in T$.

3.2 API-Call-Guided Static Recovery

Due to `ROSDiscover`'s limitation to only recover structural models, we developed an extension, called `ROSInfer` that statically infers reactive, periodic, and state-based behavior of ROS components to create a state machine of architecturally-relevant behavior.

Similar to recovering structural models, we can also made the observation that the ROS API is commonly used to implement architecturally-relevant behavior. By looking for the API calls that define callbacks for receiving a message (`ros::NodleHandle::subscribe`), sending a message (`ros::Publisher::publish`), or sleeping for the remaining time of a periodic interval (`ros::Rate::sleep`), we recover models of architecturally-relevant behavior that can then be used for model-based analysis of the system. `ROSInfer` reconstructs state machine models by identifying ROS API calls that implement these types of behavior, their argument values, and the control flow between them.

We recover reactive behavior by finding control flow from a subscriber callback to a publish call. This establishes causality between receiving a message and sending another message.

To recover periodic behavior, `ROSInfer` looks for publish calls within loops that have infinite conditions (`true` or `ros::ok`) that call sleep on a rate object. Recovering the frequency defined in the rate constructor tells lets us recover the target frequency of the periodic behavior.

To recover state-depended behavior, `ROSInfer` finds state variables, their initial values, and state transitions. Our heuristics to identify state variables are (1) the variable is used in control conditions of architecturally-relevant behavior (i.e., functions that send messages, functions that change state variables, and of their transitive callers) and (2) the variable is in global or component-wide scope, such as member variables of component classes or non-local variables. To infer the initial state (i.e., the initial values for each state variable) of the component, `ROSInfer` searches for the first definitions of the variables either in their declaration or the main method. After the state variables are identified, `ROSInfer` infers transition conditions by combining control conditions of architecturally relevant behavior using logical operators `and` and `not` depending on whether the path is taking a negation branch (e.g., the else branch of an if-statement).

We evaluated `ROSInfer` on 106 components of Autoware,[3] the world's leading open source autonomous driving software, by comparing the recovered behavior with a ground-truth obtained by manually inspecting the code and creating hand-written models of their actual behavior. If a behavior was not

3 https://www.autoware.org

found or a value not recovered, we traces this false negative back to limitations of the implementation that can be fixed with more engineering effort or limitations of the approach. We find that on our data set, the approach could recover 100 % of periodic behaviors, 84 % of reactive behaviors, 55 % of state variables, and 67 % of state transitions.

3.3 Partial-Model-Informed Dynamic Recovery

As the results from the evaluation of our current work have shown, static analysis still leaves incomplete models in some cases (e.g., due to dynamically-loaded libraries or plug-in, use of polymorphism, or values loaded at run-time. See Section 5.2 for details). Fortunately, since the models are directly derived from the source code, they could also be used to guide the creation of experiments for dynamic analysis to fill in the unknown values in incomplete models, or to identify representative paths through the system that can be used for profiling. This motivates future work on combining static and automated dynamic analysis to infer behavioral component models that contain more information about the components.

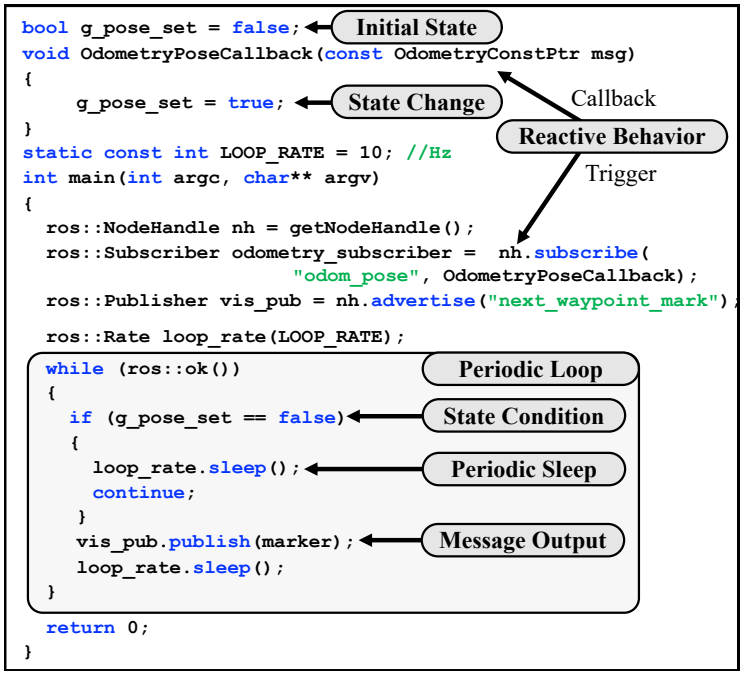
I propose to extend `ROSInfer` with dynamic analysis that automatically deploys components, systematically sends messages to it based on the known state machines to collect timing data or to resolve known unknowns.

3.4 Model-Checking of Common Properties in Robotics Systems

Combining behavioral component models with component-port-connector models, allows for analyses of intra-component-data-flow. Structural models alone do not contain information how the inputs of a component are used and what is needed for the component to produce an output. Input-output state machine models, like the ones `ROSInfer` infers, can be used to infer which messages at one component cause messages to be sent in other parts of the system. To check whether the components of a system are composed correctly, properties such as "An input at input port I_1 of component C_a can/must result in an output at output port O_1 of C_b" can be checked via discrete event simulation [15] or logical reasoning [48].

Furthermore, synchronizing the resulting component state machines at their input/output messages allows for checking arbitrary LTL properties via approaches such as PRISM [55]. Thereby safety and security properties, such as the component changing to a desired state, no messages getting lost or ignored, or a component eventually publish a certain message, can be checked [33, 30, 5].

Additionally, knowledge about the frequencies at which periodic messages get published can be used to propagate these frequencies to all transitive receivers of this data stream. Therefore, it facilitates checking the desired frequency of message publishing further down the data stream to avoid unexpectedly high publishing frequencies.



(a) Simplified example of a ROS node (lattice_trajectory_gen) that waits for an input message and then periodically publishes a message with a frequency of 10 Hz.

$$S = [g\_pose\_set : Bool]; s_0 = [g\_pose\_set = false]$$
$$M_{in} = \{odom\_pose\}; P = \{p_{10}\}$$
$$M_{out} = \{next\_waypoint\_mark\};$$
$$I = \{p_{10}, odom\_pose\}; O = \{next\_waypoint\_mark, \epsilon\}$$

Transitions:

OdometryPoseCallback($s \in S$, $i \in I$):

pre: $i == odom\_pose$

post: $g\_pose\_set' = true$ and $o = \epsilon$

periodic($s \in S$, $i \in I$):

pre: $i == p_{10} \wedge s.g\_pose\_set == true$

post: $s' = s$ and $o = next\_waypoint\_mark$

(b) Example model for code shown in Figure 3.1 (a). The first transition handles input at input port *in* and changes the state variable `ready` to true without an output. The second transition triggers periodically with a frequency of 10 Hz if the state variable `ready` is true. Then it sends a message.

Part II

Preliminary Work

4 Bug Data Set

To illustrate the variety of real-world architecture misconfiguration bugs and to foster more research and evaluation on them, we constructed and provided a data set of bugs from real-world open-source ROS systems.

This data set can be used to study properties of architecture misconfiguration bugs and to evaluate bug finding techniques for these bugs.

4.1 A Data Set of Analyzable ROS Systems

First, we identified candidate ROS systems that are viable candidates for evaluation of static architectural recovery by using the following selection criteria:

- **Programming Language:** To facilitate the evaluation of static analysis that only target C++ and can ignore Python code, we selected ROS systems primarily comprised of C++ code. For systems with a few Python components we manually created the models.
- **Availability of a Simulator:** To allow dynamic analysis and run-time observation of the behavior of ROS components, the corresponding system should come with a simulator that can be run within a Docker image.
- **Popularity:** We focused on systems that were highly starred on GitHub as representative of the target audience for `ROSDiscover`.

We used the data set from Malavolta et al. [60] as a basis for the system selection. The resulting systems are: AutoRally [32], Autoware [47], Fetch [87], Husky, and TurtleBot [79]. Table 4.1 provides an overview indicating lines of code for each system, and the number of misconfiguration bugs we considered for RQ3.

System	Stars	Lines of XML	Lines of Code	Bugs for RQ3
AutoRally	638	43 455	190 340	5
Autoware	4985	30 771	250 509	8
Fetch	126	149 664	434 022	0
Husky	264	54 699	876 405	5
TurtleBot	239	1 237 887	1 596 546	1

Table 4.1: Systems used for evaluation with their stars on GitHub (as of 8th November 2021), lines of XML configuration files, lines of code including their dependent ROS packages, and the number of bugs selected for RQ3.

System	Commits	Contributors	Releases	Bugs in Data Set
AutoRally	615	21	11	8
Autoware	3570	74	16	12
MAVROS	2503	99	40	1
Husky	511	24	46	6
TurtleBot	1142	29	92	2

Table 4.2: Statistics on the systems contained in our bug data set.

4.2 A Data Set of Architecture Misconfiguration Bugs in ROS

Second, we identified bugs within the systems described in Section 4.1 with the following methodology:

Selection Criteria: We collected documented bugs on GitHub from repositories discussed in Malavolta et al. [60] (as these repositories are well-studied and mature). For each repository, we searched for the key words "topic bug", "topic fix", "subscribe bug", "subscribe fix", "publish bug", "publish fix", "topic rename", "launch file fix", and "launch file bug" in commits, issues, and pull requests. Not all of the results refer to run-time architecture misconfigurations, and so we manually verified/filtered the bugs by inspecting the code, change history, and documentation. We excluded bugs for which we were unable to compile the software versions. As shown in Table 4.2, the data set contains 29 bugs across 5 systems. Note that this is not intended to be a complete list of architecture misconfiguration bugs in those systems.

Collected Data: For bugs caused by a broken publish-subscribe connector (i.e., inconsistent topic names or message types), we identified the publisher, topic, subscriber, and a set of launch files that launch the corresponding nodes. For bugs caused by a wrong configuration (i.e., inconsistent parameter names or parameter types), we identified the launch files and the misconfigured nodes. To formally define the misconfiguration types, we also list a corresponding architectural well-formedness rule violated by the bug. To enable users of our data set to verify their testing environment, each bug consists of a bug-commit at which the bug is present, and a bug-fix commit. Docker images for each bug containing the source code, all its dependencies, and the compiled executables for analysis can be found in the artifact.

Building Historic Project Versions: Some of the commits related to the bugs are many years old and were built with much older versions of their dependencies and much older versions of ROS (the oldest bug dates back to March 2014 and was reproduced with ROS Indigo Igloo). To support replicability, we created Docker images for each commit, each containing the versions of their build- and run-dependencies (including ROS packages, Compute Unified Device Architecture (CUDA) API version, external libraries, compilers, and the ROS distribution). If the project documented which versions of a dependencies were used, we installed these in the Docker image. Otherwise, we installed the most recent version at the time of the corresponding commit date.[4] For versions for which we were unable to construct the Docker images according to this methodology, we forward-ported the bugs (i.e., applied the bug-introducing change to a version of the software that we can build).

To identify which of these bugs can be found with structural architectural recovery, I evaluated `ROSDiscover` on this data set with the following results:

4 Using https://github.com/rosin-project/rosinstall_generator_time_machine

Bug-ID	Detected	In Theory	Description
autoware-02			Dangling connector
autoware-10			Dangling connector
autorally-01 *	✓		Inconsistent topic names
autoware-01	✓		Inconsistent topic names
autoware-04			Inconsistent topic names
autoware-05		✓	Inconsistent topic names
autoware-11	✓		Inconsistent topic names
husky-02 *	✓		Inconsistent topic names
husky-03		✓	Inconsistent topic names
husky-04 *	✓		Inconsistent topic names
husky-06 *	✓		Inconsistent topic names
autorally-05		✓	Incorrect parameter path
autorally-03 *	✓		Incorrect topic remapping
autorally-04 *	✓		Incorrect topic remapping
husky-01		✓	Incorrect topic remapping
turtlebot-01		✓	Incorrect topic remapping
autoware-03		✓	Topic name typo
autoware-09		✓	Topic name typo
autorally-02		✓	Topic name variable ignored

Table 4.3: Overview of the architectural misconfiguration bugs and whether `ROSDiscover` has detected the bug with the given rules ("Detected") or whether it is only detectable in theory due to static recovery limitations ("In Theory"). A star ("*") after the bug name means the bug was detected using forward-porting. The bug-ids reference the folders in /experiments/detection/subjects in our artifact.

4.3 Conclusions and Implications for the Proposal

This chapter has presented a newly collected data set of architecture misconfiguration bugs that will be used for evaluation of the presented approach. The following chapters will describe my current and proposed work that is intended to show we can find more architecture misconfiguration bugs than existing work (`ROSDiscover`) by inferring behavioral component models.

5 ROSInfer: Static Analysis to Infer Behavioral Component Models

This section describes our approach of static recovery of behavioral component models for ROS systems and the implementation of this approach in our tool called `ROSInfer`. Our approach is based on the observation that reactive, periodic, and state-based behavior of ROS component is often implemented using the API that ROS provides, as shown in Figure 3.1 (a). By looking for the API calls that define callbacks for receiving a message, sending a message, or sleeping for the remaining time of a periodic interval, we aim to recover models of architecturally-relevant behavior that can then be used for model-based analysis of the system.

Behavioral component models describe causal relationships between dynamic aspects of the component's interface. Each element of the behavior includes four parts:

1. An optional **output**. This describes messages being sent from the component. The formalism includes a special element for the non-output.
2. A **trigger**. In component-based systems there are three types of triggers for architecturally-relevant behavior:
   a) **Reactive triggers:** The behavior was a reaction to a message the component received at a port.
   b) **Periodic triggers:** The behavior is executed periodically. After completion it waits for the remaining time of the periodic interval so that the message gets sent with a constant frequency[5]
   c) **Component triggers:** The behavior was triggered by a component-event, such as when the component was first started, or it (un)subscribed from/to a topic.
3. **Conditions** on the state to determine whether the trigger leads to the executing the behavior.
4. **State changes** that result from the behavior.

The key idea of our approach is to reconstruct the output and the trigger of component behavior by identifying ROS API calls that implement these types of behavior, their parameter values, and the control flow between them.

5.1 Statically Inferring Component Behavior Model

The analysis process and data-flow is shown in Figure 5.1.

In general, even inferring only architecturally-relevant behavior is challenging, because theoretically any piece of code could send a message. Fortunately, the following observations about the ROS framework allow us to narrow down the analysis:

5 Note that the actual frequency can be lower than the target frequency if the component takes more time for each iteration than the target frequency allows. In this work, we ignore this case and discuss how to overcome this limitation in future work.

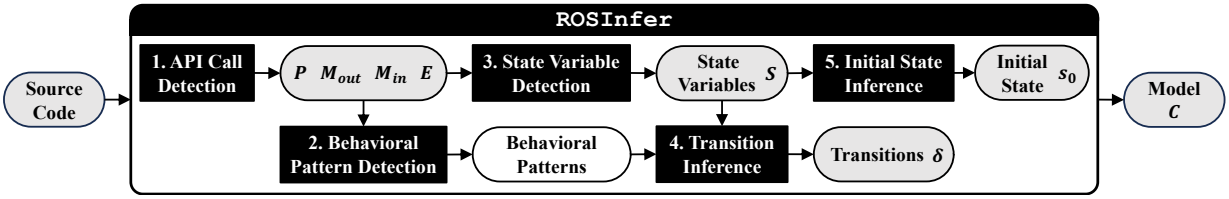


Figure 5.1: Overview of the `ROSInfer` static analysis. Boxes represent analysis steps, gray ovals represent elements of the output model, white ovals represent intermediate results, and arrows indicate data flow.

Component Framework API: Inter-component communication for sending and receiving messages happens almost exclusively via API calls that have well-understood architectural semantics [75, 82].

Behavioral Patterns Usage: The triggers of message sending behavior are usually implemented using common *behavioral patterns* (e.g., implementing periodic behavior by sending messages in an unbounded loop that sleeps for the rest of an interval).

We consider behavior to be *reactive* if it is triggered by receiving a message or a component-event (e.g., component started/stopped or (un)subscribed from/to a topic). We consider behavior to be *periodic* if it is triggered with a constant target frequency. Periodic and reactive behavior can both be *state-based*, i.e., triggered only under conditions depending on the state of the component.

The key idea of our approach is to find ROS API calls that implement the triggers or outputs of architecturally-relevant behavior, infer the API call arguments, find control flow leading to message sending behavior, and reconstruct state variables and state transitions on which architecturally-relevant behavior depends. While our approach is focusing on the ROS ecosystem, it can generalize to every framework or ecosystem for which the observations listed above hold true as well.

The remainder of the section describes each analysis step.

5.1.1 API Call Detection

The first step in API-call-guided inference of component behavioral models is to detect API calls that implement elements of architecturally-relevant behavior. `ROSInfer` accomplishes this by traversing the Abstract Syntax Tree (AST) and detecting syntactic patterns that identify architecturally-relevant API calls (see below). For most kinds of API calls, `ROSInfer` then attempts to recover the values of arguments and the object on which the function to infer additional details, such as what port owns this behavior, or the frequency / duration or sleep calls.

`ROSInfer` detects the following API calls and behaviors:

Inferring Message Outputs M_{out}: To infer message outputs M_{out} behavioral inference approaches need to identity points in the component's source code that send messages to other components. In publish-subscribe systems, this consists of API call to publish a message and corresponding wrapper APIs.

`ROSInfer` detects API calls to `Publisher::publish` and commonly used wrapper APIs (e.g., diagnostic_updater, CameraInfoManager, and `tf::TransformBroadcaster::sendTransform`). To identify the corresponding output port, `ROSInfer` infers the publisher object on which each publish call is made.

Inferring Reactive Triggers M_{in}: To infer reactive triggers, behavioral inference need to looks for the control flow entry points (i.e., callbacks that handle a received message or a requested service or the component being started). In publish-subscribe styles, subscriber callbacks define the component's behavior in response to receiving a certain message. Analogously, in call-return styles service call callbacks need to be identified.

To identify control flow entry points `ROSInfer` looks for callbacks defined as parameter to the ROS API calls `NodeHandle::subscribe` (see Figure 3.1 (a)), `MessageFilter::registerCallback`,[6] or `NodeHandle::advertiseService`.

Inferring Periodic Triggers P: To infer periodic triggers, behavioral inference needs to identity sleep calls. There are two kinds of sleep calls: (1) *constant-time sleep calls* that sleep for the same time every time they are called, (2) *filling-time sleep calls* that sleep for the remainder of a periodic interval every time they are called. Filling-time sleep calls allow the accurate static inference of the target frequency (unless the execution of each cycle takes longer than the cycle time, resulting in a lower actual frequency) while constant-time sleep calls can only provide an upper bound of the frequency, since execution times of other statements are not captured.

C++ offers three common constant-time sleep calls: `usleep`, `sleep`, and `std::this_thread::sleep_for`. The ROS framework offers `Duration::sleep`. `ROSInfer` detects these calls and infers the duration and their units from arguments using constant-folding.

ROS offers two filling-time API calls: `Rate::sleep`, which is called on a rate object (see periodic sleep in Figure 3.1 (a)), and `NodeHandle::createTimer`, which has a rate object and callback as argument. Since the frequency is specified in the constructor of the `Rate` object `ROSInfer` uses constant-folding to infer the frequency's value and denotes it with $\top$ if it cannot constant-fold it.

5.1.2 Behavioral Pattern Detection

After API call detection, `ROSInfer` builds an abstract representation of the program that contains APIs calls, control flow statements, function calls, and assignments. On this abstract representation, `ROSInfer` detects behavioral patterns that describe the architecturally-relevant behavior.

Detecting Reactive Behavior: Reactive publishing behavior is message sending that is caused by receiving a message. The pattern to detect message outputs reacting to message inputs, is checking for a path in the call graph from the callback method for each input port $in \in M_{in}$ to any of the publish calls $out \in M_{out}$. Since some systems pass publish objects as arguments to functions that then call publish on their arguments, `ROSInfer` tracks the object identity of arguments when traversing the call graph.

The other pattern for reactive behavior that `ROSInfer` identifies is if a publish call happens (transitively) within in the `main` method (and can happen in the initial state) then it responds to the component event "component-started" $\in E$.

Detecting Periodic Behavior: Periodic publishing behavior is repeated sending of a message of the same type with a constant upper target frequency (note that messages do not necessarily always have to be sent every interval). The pattern to detect periodic behavior is checking for publish calls that happen (transitively) within unbounded loops that (transitively) contain a sleep call. To identify unbounded loops `ROSInfer` considers loop conditions that are either `true` or `ros::ok()`.

6 To filter messages or to define a single callback method for multiple subscribers, ROS offers the message filters API.

5.1.3 State Variable Detection

The key idea to infer state variables statically is to look for variables in the code that store state information, such as `ready` in Figure 3.1 (a). We use the following heuristics to identify variables that represent component state.

Usage Heuristic: The variable is used in control conditions of architecturally-relevant behavior (i.e., functions that send messages, functions that change state variables, and of their transitive callers). Control conditions describe the conditions that determine whether a statement is executed.

Scope Heuristic: The variable is in global or component-wide scope, such as member variables of component classes or non-local variables. Since local variables are used close to their assignments, they are less likely to capture state information than variables that can be changed in callbacks or other functions. This heuristic limits the search space and complexity of the resulting models, because control conditions can contain complex logic that defined behavior that is not architecturally relevant.

To implement the usage heuristic `ROSInfer` first infers all control conditions for all publish calls and their transitive calls and removes conditions on variables that do not satisfy the scope heuristic and using constant folding to replace variables and constants with the literals that they represent.

5.1.4 Transition Inference

After detecting state variables are and inferring behavioral patterns for reactive and periodic behaviors, the only information that remains to be inferred to create complete transition functions δ are conditions on state variables and state changes. `ROSInfer` identifies the intra-procedural control conditions for each publish call and its transitive function calls. In an inter-procedural analysis on the call graph starting from the behavior's trigger `ROSInfer` then combines the control conditions of function calls ending in the publish call. Conditions are combined using a logical AND and negated in the case of taking the `else` branch of an if-statement.

To infer state changes, `ROSInfer` detects assignments to state variables, constant-folds the right hand-side of the assignments, and infers the assignments' triggers in the same way as for other architecturally-relevant behavior as described above. `ROSInfer` then groups behaviors by triggers and state conditions and builds the union of all outputs and state changes with the same triggers and conditions.

5.1.5 Initial Value Inference

To infer the initial state $s_o \in S$ (i.e., the initial values for each state variable) of the component, `ROSInfer` searches for the first definitions of the variables. These can be either in their declarations, in the program entry point of the component (e.g., `main`) and its transitive calls, or in statements or initializers of component class constructors. If an initial expression is found `ROSInfer` attempts to constant-fold the expression. Analogous to previous cases, values that cannot be constant-folded are denoted with $\top$.

5.2 Evaluation

In this section we describe how we evaluate the overall approach of API-call-guided recovery of component behaviors in ROS systems as well as our implementation of `ROSInfer` on large, real-world open source ROS systems.

In this section we describe how we evaluate the overall approach of API-call-guided static inference of architecturally-relevant component behavior in ROS systems as well as the results of our evaluation of ROSInfer's recovery rate, recall, precision, and execution time on five large, real-world open source ROS systems.

5.2.1 Experimental Setup

To evaluate ROSInfer we asked the following research questions:

RQ 1 (Recovery Rate) **Results in Section 5.2.2**

How high is ROSInfer's recovery rate for real-world ROS systems, i.e., what is the percentage of inferred architecturally-relevant behaviors that can be recovered completely?

When static analysis detects message sending behavior within a component's source code (e.g., a message-sending API call) it attempts to infer a complete behavioral model of what causes the component to send this message (e.g., to what input it reacts, at what periodic frequency it is sent, in what state it is sent). Since static analysis cannot always recover all parts of this behavior, resulting models can be partial (i.e., include known unknowns $\top$). To measure how often static analysis fails to infer parts of the resulting model as a measure of how complete and precise inferred models are in the practice we calculate the recovery rate for different behavior for real-world ROS components.

RQ 2 (Recall) **Results in Section 5.2.3**

How high is ROSInfer's recall for real-world ROS systems, i.e. what percentage of architecturally-relevant component behavior can ROSInfer infer correctly?

Our approach is based on the assumption that developers of ROS systems commonly use the ROS API and behavioral patterns to implement architecturally-relevant component behavior. So even if the static analysis could recover all elements of detected behaviors it might miss behaviors that violate this assumption. To validate this assumption and to evaluate how many behaviors ROSInfer missed we measured the recall compared to a ground truth. This metric measures the degree of completeness of the set of inferred behaviors on real-world ROS systems. To measure this, we executed ROSInfer on real ROS components with corresponding ground truth models and compared the output for different behavior types.

RQ 3 (Precision) **Results in Section 5.2.4**

How high is ROSInfer's precision for real-world ROS systems, i.e., what percentage of inferred architecturally-relevant component behaviors are true positives?

Since ROSInfer uses heuristics to infer architecturally-relevant behaviors, it can incorrectly classify behaviors as periodic or reactive to a component event or component input, and can include unnecessary or incorrect state variables or state transitions. To evaluate how many false positives are in the inferred models, i.e., how often ROSInfer infers behaviors that do not exist in the real program, we measured the precision of inferred models compared to a ground truth. This metric measures the degree of soundness of ROSInfer's inference heuristics on real-world ROS systems.

Overview of Evaluation Systems: For all research questions we evaluated `ROSInfer` on five large real-word open source systems: Autware.AI [47], AutoRally [32], Fetch [87], Husky, and Turtlebot [79] from the data set presented in Timperley et al. [82]. To demonstrate the complexity and size of the systems used for evaluation, statisitics are shown in **??**. Some components are part of multiple of these systems due to component reuse, leaving 542 unique components in total.

Ground Truth Models: Measuring recall and precision requires a ground truth to compare to. Unfortunately, there is no reliable ground truth available for the architectural behavior of ROS components. Therefore, we needed to create ground truth models by hand by via manual source code inspection of the five ROS systems mentioned in Table 4.1. Due to the large size and complexity of these systems, we could not construct models for all 518, so for each system we randomly picked components (excluding components that are test or demo components that do not contain architecturally-relevant behavior). Christopher Timperley and I evenly split the work.

To ensure consistency we first created a protocol for manual model inference, which is included in the replication package. The protocol includes steps to infer behaviors, a consistent format notation, and descriptions of how to handle exceptional cases that do not fit into the given format.

To validate the accuracy of manually inferred models, we measured the agreement of an overlap of 21 models (14.19 % of total) that were inferred by both authors, intentionally including some of the most complex models in this overlap. We agreed completely on 86 % of these components and partially on the remaining three components. After a discussion of the few differences in inferred models, we identified one case in which one author missed a type of publishing behavior, which resulted in revising existing models to fix their representation, and two cases of inaccurately modeled behavior that resulted in refined ground truth models.

All 155 hand-written models are also included in the replication package and are available as a data set for other researchers studying behavioral component models of ROS-based systems.

Threats to Validity: With respect to internal validity, the ground-truth models were inferred by two of the paper's authors who have not been involved in the development of the case study system. Since the creation of formal models for complex component behavior is error-prone and requires deep understanding of the domain, we cannot guarantee the correctness or completeness of all models. We attempted to reduce this threat to validity by measuring agreement between authors on model on a certain portion of handwritten models.

With respect to external validity, the results of the evaluation might not necessarily generalize to other ROS systems if their usage of the ROS API or patterns of implementing architecturally relevant behavior is significantly different from the five case study systems. We reduced this treat by selecting diverse case studies with Autoware and AutoRally being mostly self-contained industrially-developed system and Husky, Fetch, and Turtlebot following a typical open-source mentality.

5.2.2 Measuring Recovery Rate (RQ1)

Methodology: As discussed in Section 5.1, `ROSInfer` denotes values that cannot be statically recovered with $\top$ to indicate unknown values. So the main metric for the recovery rate is how often $\top$ is included in parts of the resulting model.

We ran `ROSInfer` on all 542 components of the five systems presented in **??**. Components that are includes in multiple systems only count once. For 12 components the static analysis crashed due to errors in Clang, 7 timed out after 1 hour, so these components are excluded from the evaluation, leaving 518. For

Table 5.1: Results for RQ1: The trigger types recovery rate is the percentage of inferred publish calls for which ROSInfer can infer what kind of trigger causes that behavior (periodic or reactive). For each sub-type of behavior, percentages show how many of that type do not contain unknowns (⊤) in the inferred modes of a total of 518 components of the five large real-world systems presented in **??**. *N* is the total number of behaviors of the respective type inferred by ROSInfer (all publish calls in the case of trigger types). The All row counts components that are included in multiple systems only once.

System	Trigger Types	Periodic Rates	Reactive Triggers	Initial States	State Changes
AutoRally	68.42% ($N = 38$)	78.57% ($N = 14$)	100.00% ($N = 12$)	100.00% ($N = 1$)	100.00% ($N = 1$)
Autoware	90.24% ($N = 420$)	93.88% ($N = 98$)	100.00% ($N = 281$)	73.61% ($N = 72$)	80.90% ($N = 199$)
Fetch	86.21% ($N = 29$)	0.00% ($N = 1$)	100.00% ($N = 24$)	66.67% ($N = 6$)	100.00% ($N = 5$)
Husky	92.45% ($N = 53$)	90.91% ($N = 11$)	100.00% ($N = 38$)	45.45% ($N = 11$)	100.00% ($N = 9$)
Turtlebot	88.10% ($N = 42$)	66.67% ($N = 12$)	100.00% ($N = 25$)	80.77% ($N = 26$)	100.00% ($N = 19$)
All	87.87% (**$N = 544$**)	90.23% (**$N = 133$**)	100.00% (**$N = 345$**)	76.70% (**$N = 103$**)	82.96% (**$N = 223$**)

Table 5.2: Recall and precision of ROSInfer based a comparison with 148 manually inferred component models. *TP*, *FP*, and *FN* are the number of true positives, false positives, and false negatives compared to the ground truth models.

System	Models	Periodic Behaviors			Reactive Behaviors			State Variables			State Transitions		
		TP	*FN*	*FP*	*TP*	*FN*	*FP*	*TP*	*FN*	*FP*	*TP*	*FN*	*FP*
AutoRally	13	15	0	0	11	6	0	1	2	0	1	3	0
Autoware	119	22	2	0	147	18	15	30	10	7	43	12	4
Fetch	11	1	0	0	8	5	0	3	0	0	2	1	0
Turtlebot	5	2	0	0	5	0	2	2	0	1	3	0	0
All	148	40	2	0	171	29	17	36	12	8	49	16	4
Recall		95.2 % (of **42**)			85.5 % (of **200**)			75.0 % (of **48**)			75.4 % (of **65**)		
Precision		100.0 % (of **40**)			91.0 % (of **188**)			81.8 % (of **44**)			92.5 % (of **53**)		

each type of architectural behavior we then calculated the percentage of unknowns included in inferred values (i.e., target frequencies for periodic behavior, triggering events or callbacks for reactive behavior, initial values for state variables, and new values for state transitions). These numbers represent how well ROSInfer can infer all parameters of a detected behavior.

Further, the *trigger types recovery rate* metric measures how often ROSInfer can recover the trigger for detected publishing behavior:

Trigger Types Recovery Rate (Evaluation Metric)

The *trigger types recovery rate* approximates the inferred proportion of the total architecturally-relevant component behavior by measuring the percentage of message publishing calls for which ROSInfer can infer the cause of the behavior (i.e., for which a behavioral pattern with corresponding trigger was detected).

Note that this metric overapproximates recall in cases in which publish calls are hidden in inaccessible source code (e.g., in DLLs) but underapproximates recall in cases in which publish calls happen in uncalled callback (e.g., XbeeCoordinator and obstacle_sim).

After the quantitative analysis, we manually inspected each case of unknown values to conduct an in-depth qualitative analysis of the limitations of `ROSInfer` using open coding and linked examples.

Results for RQ 1 (Recovery Rate) **See Table 5.1**

In a exhaustive analysis of five large real-world ROS systems with 518 components **the overall trigger types recovery rate is** 88 %. The proportion of inferable values is 90 % for periodic rates, 100 % for reactive triggers, 77 % for state variable initial values, and 83 % for state changes.

Results: Detailed quantitative results are shown in Table 5.1.

AutoRally has the lowest trigger types recovery rate, because many components respond to inputs of serial devices with **project-specific API** (e.g., AutoRallyChassis, GPSHemisphere).

Cases in which `ROSInfer` cannot recover periodic rates include rates that are loaded from **component parameters** (e.g., runStop, fake_camera, adis16470_node, robot_pose_ekf, yocs_virtual_sensor, lidar_fake_perception, AutoRallyChassis, watchdog_node), **return values of function calls** (e.g., robot_pose_ekf), **conditional behavior** (e.g., yocs_virtual_sensor).

Cases in which `ROSInfer` cannot recover initial states include primitive types with **implicit initialization** (e.g., decision_maker_node), **ignored functions** (e.g., tl_switch, decision_maker_node, way_planner_core).

State transitions include unknowns if and only if the right hand-side of assignments or state variables that cannot be constant-folded.

Reactive triggers can be recovered completely, since `ROSInfer`'s current implementation does not include component events or message inputs that can include unknown values.

5.2.3 Measuring Recall (RQ2)

Methodology: After creating the handwritten models as ground truth (see Section 5.2.1) we executed `ROSInfer` on the source code and compared the results by treating the handwritten models as ground truth. The existence of model elements is compared automatically, while expressions in conditions are compared by humans to judge whether they are logically equivalent. After the quantitative analysis, we then manually inspected each false negative to conduct a qualitative root cause analysis of missed behaviors.

Results for RQ 2 (Recall) **See Table 5.2**

In a ground-truth comparison with 148 components `ROSInfer` has a recall of 95 % for periodic behavior, 86 % for reactive behavior, 75 % for state variables, and 75 % for state transitions.

Results: Detailed quantitative results are shown in Table 5.2.

Cases in which `ROSInfer` cannot detect reactive behavior include the use of **virtual methods** (e.g., joystick_teleop), behavior that is **triggered by other events** than receiving a message in subscriber callback, such as reacting to messages from external devices received via serial ports (e.g., vg440_node), Mqtt messages (e.g., mqtt_receiver), or CAN-Bus (e.g., vehicle_receiver), our approach cannot infer the trigger for this behavior.

Cases in which ROSInfer cannot recover state variables include **complicated object logic**, such as whether a list or map is empty (e.g., vscan2image). Figure 5.2 shows an example of this. This requires a deeper understanding of the objects owned by the component that are used to represent its state and are therefore a limitation of the approach. Conditions on object fields can contain implicit dependencies that cannot easily be inferred statically. For example when a subscriber callback initializes the image stored in a state variable whose width and height are checked to be positive numbers in a control condition (`image.width > 0 && image.height > 0`) a human developer can infer that this condition refers to checking whether the initialization in the subscriber callback has been called implying that the component has received the message. This dependency that is implicit due to complex logic within the imagine object cannot be inferred statically.

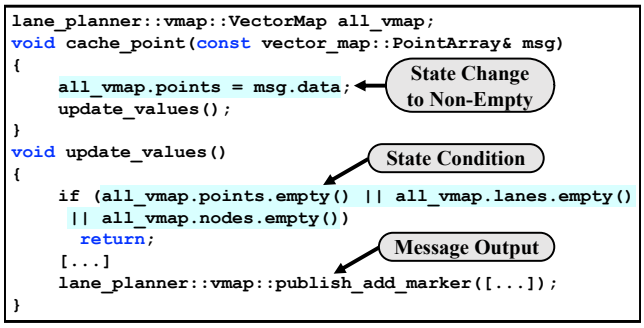


Figure 5.2: Simplified code snippet showing an example from waypoint_clicker in Autoware.AI for which our approach cannot recover the state machine. The analysis would need to model the state of a vector map containing multiple arrays and identify that the assignment in the `cache_point` subscriber callback affects the return value of the `empty` call.

5.2.4 Measuring Precision (RQ3)

Methodology: For each behavior category we calculated the number of inferred behaviors that are part of the output of ROSInfer but not part of the ground truth models. We then manually inspected each false positive to conduct a qualitative root cause analysis of incorrectly classified behaviors.

Results for RQ 3 (Precision) **See Table 5.2**

In a ground-truth comparison with 148 components ROSInfer has an precision of 100 % for periodic behavior, 91 % for reactive behavior, 82 % for state variables, and 92 % for state transitions.

Results: Detailed quantitative results are shown in Table 5.2.

False positives for reactive behavior are caused by a limitation of our current implementation that treats periodic behavior in main as reactive to component-started regardless of potential state-conditions, which can be fixed in the future.

False positives of state variables are caused by mistaking a **configuration parameter** for a state variable (e.g., amcl), mistaking **variable identity** due to overloaded variable names (e.g., control dependencies on assignments to **another state variable false positive** (e.g., pos_downloader), and control dependencies on assignments to **another state variable false positive** (e.g., pos_downloader).

False positives of state transitions are caused by false positives of the corresponding state variables.

5.2.5 Measuring Execution Time

When running ROSInfer on Autoware.AI on a server with 4 Intel(R) Xeon(R) Gold 6240 CPUs (each has 18 cores at 2.60 GHz) with 256 GB RAM, the static analysis took on average 44.93 s. The fully automated

analysis of the entire Autoware.AI system took 4.58 h and much shorter for the other systems (Autorally: 18.23 min, Fetch: 31.25 min, Husky: 35.28 min, Turtlebot: 49.40 min). This should demonstrate that the static analysis scales to real-world systems and could integrate well into iterative software development practices.

In practice, static model inference approaches like the presented approach would integrate well in iterative development processes since they support automatic regeneration of models when they sources change. Changes to the code base require regeneration of only the components that are affected by the code change, since `ROSInfer` infers which source files are required to infer each component model. This would dramatically reduce the time to update the system's behavioral model.

The effort it took to create the 155 handwritten models of this evaluation can be approximated with about 120 work hours of manual labor. In practice, the developer time saved will be lower than the difference between these two numbers, because developers potentially need to replace the known unknowns ($\top$) with correct values and cannot fully rely on the inferred models being complete. While in this paper we do not quantify the saved effort, we present these numbers to demonstrate that the approach can save a significant portion of time to infer models, making model-based analysis more accessible, economical, and scalable to large systems.

5.3 Conclusions and Implications for the Proposal

This chapter has shown additional evidence towards the thesis statement by showing that assumptions about framework-specific APIs enable the automatic inference of partial behavioral component models for ROS-based robotics systems. The following chapters will describe how I propose to complete the partial models and how to show that these models can be practically useful to find architecture reconfiguration bugs.

Part III

Proposed Work

6 Completion of Behavioral Models using Dynamic Analysis

As the results from the evaluation of `ROSDiscover` and `ROSInfer` have shown even perfect static analysis still leaves incomplete models in some cases. Fortunately, since the models are directly derived from the source code, they could also be used to guide the creation of experiments for dynamic analysis to fill in the unknown values ($\top$) in incomplete models, or to identify representative paths through the system that can be used for profiling. This motivates future work on combining static and automated dynamic analysis to infer behavioral component models that fill in the known unknowns of the statically inferred models.

I propose to extent `ROSInfer` with dynamic analysis that automatically deploys components, systematically sends messages to it based on the known state machines to collect data that resolves known unknowns.

6.1 Existing Work

Existing work from other software domains [16, 17, 40] controls the system-level inputs and observes the behavior of a running system to extract behavioral models. The lack of controllability of component-level inputs of system-level approaches result in not being able to observe traces through components that would be caused by a changes in collaborating components. Therefore, resulting component models cannot predict how the component behaves under changes to its collaborating components or when the component is reused in a different system, which is an important requirement for behavioral component models to be useful within the reuse-heavy ROS ecosystem.

First, the analysis will instrument the code of each component based on the known unknowns of the partial, statically-inferred models. Second, the analysis will deploy the components in the system in a configuration found in existing launch files. Then, the analysis will observe the components' behavior while controlled the input ports to systematically cover a large portion of the component's interface. Finally, the individual component behavior models get combined based on a given architecture configuration to analyze the system behavior. Since the component behavior models capture a wider range of component environments than those visible in the configuration of the current design, the resulting models can be used to predict the system-level behavior under changes to the system to find better architectural configurations or simulate a change to the architecture before implementing it.

6.2 Approach

6.2.1 ROS Component Testbed Infrastructure

To address the challenges of controllability and observability of component behavior, I propose to develop a component testbed instrastructore for ROS components.

Deploying Components: To deploy each component, we will identify launch files from repository

that represent an complete configuration of the system. This step requires available launch files for each node. If they cannot be found in the project directly, they need to be specified before-hand.

Controlling Components via Mocked Inputs: Due to the decoupled nature of publish-subscribe connectors, sending inputs into ROS components is generally easier than in the general case, as it only requires to publish a message to the input topics of a component. ROS offers a replay `rosbag` files to send timed messages to topics. If `rosbag` turns out to be not flexible enough for this purpose, an alternative options is to generate a small script that sends messages to topics and sleeps for the durations between messages. Service calls and action calls need programmatic triggers via generated script. Since the interface of the ports is known due to `ROSDiscover` [82], all information needed to generate API calls is available.

Observing Internal Component Behavior via Automated Application Instrumentation: To observe internal component behavior that cannot be recovered statically, which results in unknown values $\top$ in statically recovered models, I propose to instrument the application code with additional logging functionality. Since the recovered models from `ROSInfer` map model elements to source code locations each unknown model value $\top$ can be traced precisely to where in the code it is being used. Therefore, automatic code instrumentation can add logging code in these locations to write dynamically observed values for unknown model values into a separate logging file.

6.2.2 Experiment Creation

An important research challenge for this contribution is finding a wide range of representative inputs and and environment configurations without requiring exhaustive testing. For this purpose, the availability of path conditions to architecture-relevant behavior inferred via static analysis could be helpful. Also, existing research in fuzzing can support systematic exploration of the input space.

6.2.3 Model Inference

The final step of the dynamic analysis is to aggregate observations to extend statically-inferred component models. This step involves automatically identifying clusters of values and grouping them by a unifying condition. I plan to use existing data analysis techniques for this step.

6.3 Evaluation

The evaluation of completing statically inferred models will follow the same research questions, use the same metrics, and use the same data as the evaluation of `ROSInfer` (see Section 5.2).

6.4 Conclusions and Implications for the Proposal

This chapter has outlined my plan to complete the partial models generated with dynamic analysis. The following chapters will describe how I propose to show that these models can be practically useful to find architecture reconfiguration bugs to provide further evidence for the thesis statement.

7 Model-based Analyses for Robots

After presenting my approach to infer behavioral component models, this chapter describes my proposed work on creating model-based analyses to find bugs based on these models to demonstrate that the models are useful.

7.1 Proposed Analyses

The models inferred with the proposed approach can be analyzed for a variety of properties and behaviors. This section lists the properties and behaviors that could be supported, discusses the implications of supporting them, estimates the effort to do so, and prioritizes them based on estimated effort and importance.

Deadlock Freedom: To avoid a situation in which components get into a deadlock because they indefinitely wait for each other's messages model-based analyses can check for deadlocks in the system. The are two types of deadlock analyses: (1) *system-wide deadlock*: the entire system stops making pro-0gress, (2) *component-wide deadlock*: A single component or set of components cannot make progress. An analysis for system-wide deadlocks is built into TLA+ and can be activated with a simple check box. Since some components intentionally remain in a finial state, component-wide deadlock analysis requires the user to specify a list of components that should not have a deadlock. Based on this, we can generate an LTL property to check for deadlocks. Generating properties for component-wide deadlock freedom has a medium priority for this thesis proposal.

Liveness Properties: To ensure that a certain type of desired system behavior happens, such as "eventually the planning component reaches the ready state", liveness property verification is needed. Since component states are specified directly in the TLA+ model, liveness properties can be specified directly in TLC, e.g., "`<>A_ready = true`". Eventually sending a message can be specified with the message queue of the output port being non-empty, e.g., "`<>(c_out /= «»)`". To specify that behavior can always happen (again) after any given point in the execution of the system, the operator for always eventually "`[]<>`" can be used. Since TLA+ allows to verify arbitrary LTL properties via TLC and there will be a large variance of liveness properties to specify, we propose that users would specify their properties directly in TLC and using built-in verifiers to check them. The necessary analysis capabilities for these properties are implemented in TLA+ already and they do not require adjustments to the format of the models.

Causal Reaction Properties: To prevent bugs such as missing or inconsistent connectors, lost messages, or ignored inputs it is desirable to check properties that verify a causal relationship between an action and its reaction, such as a input A results in eventually sending a response B. The corresponding LTL property for this type of behavior is `[](A => <>B)`. TLC offers easier syntax for this: `A ~>B`. Using this type of syntax properties such as message A results in message B can be specified as: `a_out /= «» ~> c_in /= «»`. The necessary analysis capabilities for these properties are implemented in TLA+ already and they do not require adjustments to the format of the models.

7.2 Generation of PlusCal/TLA+ Models

Based on the inferred state machine models I propose to generate analyzable PlusCal/TLA+ models, such as the one shown in Listing B.1.

Components are modeled as fair processes with each transition receiving a labeled action that contains an if-statement with with the transition's pre condition and the post conditions in the body. Input queues and output queues are modeled as lists that are constrained in their length. Topics are modeled as processes that take elements from the output queues and add them to all input queues of subscribers.

To model reactive triggers the pre-condition checks for the input queue being empty:

```
if base_waypoints_lattice_trajectory_gen ≠ <> then
```

Periodic triggers are modeled as action that do not have any preconditions besides the conditions state and input conditions inferred by `ROSInfer`.

Ports are modeled as sequences of messages from which the component pulls messages when responding to them as topic processes adding messages when a publisher published to the topic.

State variables are modeled as variables maintained by the process that models the corresponding component and assigned to their initial values in the variable declaration.

7.3 Conclusions and Implications for the Proposal

This chapter has outlined my plan to show how statically and dynamically inferred component behavioral models can be used to automatically find potential architecture misconfiguration bugs.c The last chapter will describe how I propose to evaluate the effectiveness and practical usefulness of the overall approach to provide the finial evidence for the thesis statement.

8 End-To-End Evaluation

The goal of the end-to-end evaluation is to answer the following two high-level research questions:

RQ 1 (Effectiveness)	**Methods in Section 8.1**
How **effective** is `ROSInfer` at finding **realistic source-level behavioral architecture composition bugs**?	

The static model inference of `ROSInfer` has been evaluated in Section 5.2 and an evaluation proposal for dynamic inference is outlined in Section 6.3. This research question closes the gap between individual pieces of `ROSInfer` to evaluate its end-to-end ability to find bugs in the source code. I propose to answer this research question by injecting realistic bugs into the source code and checking how many can be found in the inferred models. More details on the methodology is described in Section 8.1.

RQ 2 (Practicality)	**Methods in Section 8.2**
How **useful** is `ROSInfer` in a **practical setting**?	

I propose to answer this research question by comparing the performance of `ROSInfer` to typical ROS developers. More details on the methodology is described in Section 8.2.

8.1 Measuring Effectiveness

To evaluate the effectiveness of finding bugs I propose to inject realistic bugs into the source code of real-world ROS systems, run `ROSInfer`, TLA+ generation, and model-based analyses to identity potential bugs, and compare the detected bugs with the ground truth.

Creating a Data Set of Real-World Bugs: Since there is no existing data set of behavioral architecture composition bugs in ROS, I will need to create one as part of the research contribution. The first step in this effort is to **find existing bugs** in open source systems. Analogously to the data set creation for the evaluation of `ROSDiscover` described in Section 4.2, I will search GitHub issues, pull requests, and commit messages for keywords that might indicate behavioral architecture composition bugs. The terminology for these keywords can come from ROBUST [83], Paulo's paper on ROSAnswers, conversations with ROS developers, and search for other architectural terms, such as "indefinite waiting", "fix mandatory input", "race condition", "state bug", "wrong state", "wrong frequency", "unexpected rate", "fix rate", "can't keep up", "message loss", "deadlock", "fix queue size", and "communication bug" as well as variations of these. This data set will be published as artifact for further researchers to benchmark their approaches to find behavioral architecture composition bugs.

So far, three bugs from the `ROSDiscover` data set are included in this data set.

Bug Injection: In the case that the data set of real-world bugs is not large enough and/or does not cover a large enough variety of behavioral architecture composition bugs, I propose to manually inject realistic bugs into the source code of real-world ROS systems. Real-world bugs are preferred, since they

minimize the potential threat to validity of over-fitting and biasing the evaluation to implementation patterns and specific instances of behavioral architecture composition bugs that ROSInfer was designed to find. Therefore, bug inject will only be a back-up.

Data Analysis: To analyze the resulting data, I propose to quantitatively measure **precision** and **recall** as both false positives and false negatives impact how effective ROSInfer can be in a practical setting. Additionally, a **qualitative root cause analysis** of the kinds of false positives and false negatives (analogous to the evaluation of ROSInfer described in Section 5.2) will provide further insights into limitations and potential future work to improve the effectiveness of the approach.

8.2 Measuring Practicality

8.2.1 Sub-Research Questions

To measure how useful ROSInfer is in a practical setting, I propose to answer the following sub-research questions:

RQ 2.1 (Practicality)

How does ROSInfer 's **effectiveness** compare to typical ROS developers in finding bugs?

This questions evaluates whether ROSInfer can find more or less bugs than a typical ROS developer.

RQ 2.2 (Practicality)

How does ROSInfer 's **efficiency** compare to typical ROS developers in finding bugs?

RQ 2.3 (Practicality)

How useful to typical developers perceive mROSInfer to be in their development activities?

8.2.2 Setup

Participants will be provided with a git repository containing a real-world ROS system with one or more bugs from the data set described in Section 8.1.

Then, we will ask the participants to **list all architecture misconfiguration bugs** that they can find in the system while **measuring the time** it takes them to come to a conclusion.

Then we show them the output of the tool, ask them to locate the bug and ask for a description how they would fix it (without having to code it).

A final survey will ask them questions about their experience, usefulness of the output, and importance of the problem.

8.2.3 Recruiting

To recruit participants, I plan to ask master students in the RI master programs, pro-actively contact attendees of at robotics conferences and developers in ROS-companies, such as NREC, Siemens, or PAL Robotics and advertise the study on social media platforms.

Pre-existing knowledge for the study includes experience in building medium-size or large-size ROS 1 systems in C++, understanding of architectural composition, and basic understanding of architecture-level bugs. The pre-required knowledge can be assessed with a short quiz of ROS concepts and behavioral architecture composition bugs in minimal examples.

The study will need to compensate participants for the time spent in the form of Amazon git cards of a value of around 20–40 USD per hour to value the expertise required for answering the survey.

8.2.4 Data Analysis

To answer RQ 2.1, I propose to measure the **precision** and **recall** of participants' answers and compare them to `ROSInfer` from RQ 1 as a way to compare the effectiveness.

Further, to get a rough estimate of the time comparison, I propose to report the time it took participants to answer the questions. This metric is likely noisy and dependent on the expertise of participants. Therefore, the order of magnitude of the difference is more important than the actual values.

9 Timeline

I plan to finish dynamic inference by the end of Summer 2024, PlusCal/TLA+ generation by the end of Fall 2024, and the end-to-end evaluation by the end of Spring 2025, so that I can defend by the end of summer 2025.

Part IV

Appendix

A ROS API Calls

This chapter describes architecturally-relevant APIs in ROS for reference.

Subscribe Call: `ros::NodeHandle::subscribe(...)`

The subscribe call defines a subscriber port. It creates and returns the newly created Subscriber object.

Advertise Call: `ros::NodeHandle::advertise(...)`

The advertise call defines a publisher port. It creates and returns the newly created a Publisher object.

Publish Call: `ros::Publisher::publish(...)`

The publish call implements the behavior of sending a message via publish-subscribe in ROS. It is called on a Publisher object. Hence, to identify the output port at which the message is sent, the advertise call that creates the Publisher object needs to be considered.

In some ROS nodes that are designed for dynamic configurations, publish calls can be called on method parameters, requiring to track the call arguments. However, in most cases, publish is called directly on the constructed objects.

Sleep Calls

There are two kinds of sleep calls: (1) *constant-time sleep calls* that sleep for the same amount of time every time they are called, (2) *filling-time sleep calls* that sleep for the remainder of a periodic interval every time they are called. Filling-time sleep calls allow the accurate static inference of the target frequency (unless the execution of each cycle takes longer than the cycle time, resulting in a lower actual frequency) while constant-time sleep calls can provide only an upper bound on the frequency, since execution times of other statements are not captured.

C++, the most commonly used programming language by ROS projects, offers three common constant-time sleep calls: `usleep`, `sleep`, and `std::this_thread::sleep_for`. The ROS framework offers `ros::Duration::sleep`.

ROS offers two filling-time sleep calls: `ros::NodeHandle::createTimer`, which has a rate object and callback as argument. The frequency is specified in the constructor of the `Rate`.

OK Call: `ros::ok()`

The API Call `ros::ok()` checks the status of the component. During normal operation of the node calls to `ros::ok()` always return `true`. It returns `false` if and only if the node has been shutdown, signaling that it should stop all ongoing computation. Therefore, it is often used in conditions of periodic behavior to stop loops that would otherwise be endless loops.

B Example Models

```
------------------------------ MODULE autoware10 ------------------------------
EXTENDS Sequences, Integers, TLC, FiniteSets
CONSTANTS Obj_reproj, Image_obj_tracked, Current_pose, Unknown, Data, NULL,
    MaxQueue

ASSUME NULL ∉ Data


\* helper functions
SeqOf(set, n) == UNION {[1..m -> set] : m ∈ 0..n} \* generates all sequences no
    longer than n consisting of elements in set
seq ⊕ elem == Append(seq, elem)

(*--fair algorithm polling
variables
image_obj_tracked = <>;
image_obj_tracked_obj_reproj = <>;
current_pose = <>;
current_pose_obj_reproj = <>;
obj_label_marker = <>;
obj_label = <>;
unknown = <>;
unknown_obj_reproj = <>;
obj_label_bounding_box = <>;

define
TypeInvariant ==
image_obj_tracked ∈ SeqOf(Data, MaxQueue) ∧
image_obj_tracked_obj_reproj ∈ SeqOf(Data, MaxQueue) ∧
current_pose ∈ SeqOf(Data, MaxQueue) ∧
current_pose_obj_reproj ∈ SeqOf(Data, MaxQueue) ∧
obj_label_marker ∈ SeqOf(Data, MaxQueue) ∧
obj_label ∈ SeqOf(Data, MaxQueue) ∧
unknown ∈ SeqOf(Data, MaxQueue) ∧
unknown_obj_reproj ∈ SeqOf(Data, MaxQueue) ∧
obj_label_bounding_box ∈ SeqOf(Data, MaxQueue)
```

```
Response == <>(obj_label_marker ≠ <>)
end define;

fair process obj_reproj ∈ Obj_reproj
variables
msg ∈ Data;
isReady_ndt_pose = FALSE;
isReady_obj_pos_xyz = FALSE;
ready_ = FALSE;

begin

image_obj_tracked_obj_reproj:
if image_obj_tracked_obj_reproj ≠ <> then
msg := Head(image_obj_tracked_obj_reproj);
image_obj_tracked_obj_reproj := Tail(image_obj_tracked_obj_reproj);
obj_label := obj_label ⊕ msg;
obj_label_marker := obj_label_marker ⊕ msg;
obj_label_bounding_box := obj_label_bounding_box ⊕ msg;

if ((ready_ ∧ isReady_obj_pos_xyz) ∧ (isReady_obj_pos_xyz ∧ isReady_ndt_pose))
    then
isReady_ndt_pose := FALSE;
isReady_obj_pos_xyz := FALSE;


elsif ready_ then
isReady_obj_pos_xyz := TRUE;


end if;

end if;

current_pose_obj_reproj:
if current_pose_obj_reproj ≠ <> then
msg := Head(current_pose_obj_reproj);
current_pose_obj_reproj := Tail(current_pose_obj_reproj);
obj_label := obj_label ⊕ msg;
obj_label_marker := obj_label_marker ⊕ msg;
obj_label_bounding_box := obj_label_bounding_box ⊕ msg;

if (isReady_obj_pos_xyz ∧ (isReady_obj_pos_xyz ∧ isReady_ndt_pose)) then
```

```
isReady_obj_pos_xyz := FALSE;
isReady_ndt_pose := FALSE;


elsif TRUE then
isReady_ndt_pose := TRUE;


end if;

end if;

unknown_obj_reproj:
if unknown_obj_reproj ≠ <> then
msg := Head(unknown_obj_reproj);
unknown_obj_reproj := Tail(unknown_obj_reproj);

if TRUE then
ready_ := TRUE;


end if;

end if;

end process;

fair process image_obj_tracked ∈ Image_obj_tracked

begin
Write:
if image_obj_tracked ≠ <> then
msg := Head(image_obj_tracked);
image_obj_tracked := Tail(image_obj_tracked);
image_obj_tracked_obj_reproj := image_obj_tracked_obj_reproj ⊕ msg;

end if;
end process;

fair process current_pose ∈ Current_pose

begin
Write:
if current_pose ≠ <> then
```

```
msg := Head(current_pose);
current_pose := Tail(current_pose);
current_pose_obj_reproj := current_pose_obj_reproj ⊕ msg;

end if;
end process;

fair process unknown ∈ Unknown

begin
Write:
if unknown ≠ <> then
msg := Head(unknown);
unknown := Tail(unknown);
unknown_obj_reproj := unknown_obj_reproj ⊕ msg;

end if;
end process;

end algorithm; *)
====
```

Listing B.1: Example TLA+ Code

Glossary

Symbols

Rosbag: A tool that allows to record and replay messages published to topics in ROS. See http://wiki.ros.org/rosbag.
Pages: 36

A

API: An Application Programming Interface (API) is a set of specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API.
Pages: iii, 4, 5, 9, 14, 20, 23, 24, 26, 36, 47

AST: An Abstract Syntax Tree (AST) is a representation of source code that hierarchically splits programing language constructs into parent-child relationships.
Pages: 24

B

Behavioral model: A behavioral model is an instance of the behavioral view. See Section 2.1.3.
Pages: 3, 4, 6

Behavioral view: The behavioral view, also known as dynamic view, expresses the behavior of the system or its parts. It can describe input-output relationships for components, states, state transitions, actions, and other behavioral properties such as timing. See Section 2.1.3.
Pages: 3, 7, 8, 10, 53

C

Component: A component is an independently deployable run-time unit of software (e.g., processes) that communicates with other components primarily via messages. See Section 2.1.2.
Pages: 7–11, 23–32, 35, 36, 38, 53, 54

Component-connector model: A component-connector model is an instance of the component-connector view. See Section 2.1.2.
Pages: 3, 10, 13

Component-connector view: The component-connector view, also known as run-time architecture, represents a structural configuration of the architecture at run time containing components, connectors, and ports. See Section 2.1.2.
Pages: 7, 10, 53

Connector: . See Section 2.1.2.
Pages: 8, 36, 53

Constant-time sleep call: A constant-time sleep call sleeps the thread for the same amount of time every time it is are called. See Appendix A for examples..
Pages: 47

Controllability: In dynamic analysis of software, *controllability* describes the degree of effort needed to re-create the execution conditions that should be analyzed via by controlling inputs, state, and hardware conditions [28].
Pages: 35

CPU: Central Processing Unit.
Pages: 54

CUDA: Compute Unified Device Architecture (CUDA) is a Graphics Processing Unit (GPU)-based parallel computing framework and API.
Pages: 20

D

Deployment view: The deployment view, also known as physical view, describes the hardware configuration of a system. It maps components to processing units (e.g., Central Processing Units (CPUs), GPUs, servers) and shows how processing units connected with each other via hardware networks. See **??**.
Pages: 10

F

Filling-time sleep call: A filling-time sleep calls sleeps the thread for the remainder of a periodic interval every time they are called. See Appendix A for examples..
Pages: 47

G

GPU: Graphics Processing Unit.
Pages: 54

L

LTL: Linear Temporal Logic.
Pages: 11, 15

M

Module view: The module view, also known as code view, displays the software the way programmers interact with the source code. It contains source code elements, such as packages, classes, methods, or data entries and their relationships. See Section 2.1.1.
Pages: 7

N

Node: A node is a component in ROS. See Section 2.4.1.
Pages: 9, 10, 16, 20, 36, 47

O

Observability: In dynamic analysis of software, *observability* describes the degree of effort needed to take measurements of a running programs, such as direct outputs, indirect outputs, intermediate exeuction steps, and quality attributes [28].
Pages: 35

P

Port: . See Section 2.1.2.
Pages: 3, 5, 35, 36, 38, 47, 53

Publish-subscribe: Publish-Subscribe is an asynchronous message sending connector that loosely couples senders (i.e., *publishers*) from receivers (i.e., *subscribers*) via a know intermediary interface. See Section 2.2.1.
Pages: 8, 9, 36, 47, 55

Publisher: A publisher is the role of the sender in the publish-subscribe style. It sends messages to topics that forward them to every component that subscribed before the message was sent. See Section 2.2.1.
Pages: 8, 20, 38, 47, 55

R

ROS: The Robot Operating System (ROS) is the most popular open-source framework for component-based robotics systems. See Section 2.4.
Pages: iii, 3–6, 9, 10, 14, 16, 19, 20, 23, 26, 35, 36, 47

S

Subscriber: A subscriber is the role of the receiver in the publish-subscribe style. It after it subscribed to a topics it receives all messages sent by the corresponding publishers. See Section 2.2.1.
Pages: 8, 20, 47, 55

T

Topic: Topics is ROS implementations of the publish-subscribe style. They are represented as strings. See Section 2.4.2.
Pages: 20, 23, 38, 55

U

UML: The Unified Modeling Language (UML) is a general-purpose modeling language that is intended to provide a standard way to visualize the design of a system.
Pages: 7

Bibliography

[1] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher S. Timperley. 2020. **A Study on Challenges of Testing Robotic Systems**. In *International Conference on Software Testing, Validation and Verification (ICST '20)*, 96–107. DOI: 10.1109/ICST46399.2020.00020.

[2] Afsoon Afzal, Deborah S. Katz, Claire Le Goues, and Christopher S. Timperley. 2021. **Simulation for Robotics Test Automation: Developer Perspectives**. In *Conference on Software Testing, Verification and Validation (ICST '14)*. IEEE, 263–274. DOI: 10.1109/ICST49551.2021.00036.

[3] Aakash Ahmad and Muhammad Ali Babar. 2016. **Software Architectures for Robotic Systems: A Systematic Mapping Study**. *Journal of Systems and Software*, 122, (December 2016), 16–39. DOI: 10.1016/j.jss.2016.08.039.

[4] Michel Albonico, Milica Đorđević, Engel Hamer, and Ivano Malavolta. 2023. **Software engineering research on the robot operating system: a systematic mapping study**. *Journal of Systems and Software*, 197, 111574. DOI: https://doi.org/10.1016/j.jss.2022.111574.

[5] Nikolaos Alexiou, Stylianos Basagiannis, and Sophia Petridou. 2016. **Formal security analysis of near field communication using model checking**. *Computers & Security*, 60, 1–14. DOI: 10.1016/j.cose.2016.03.002.

[6] Periklis Andritsos, Panayiotis Tsaparas, Renée J. Miller, and Kenneth C. Sevcik. 2004. **LIMBO: Scalable Clustering of Categorical Data**. In *International Conference on Extending Database Technology (EDBT '04) - Advances in Database Technology*. Springer, 123–146. DOI: 10.1007/978-3-540-24741-8_9.

[7] Hugo Araujo, Mohammad Reza Mousavi, and Mahsa Varshosaz. 2023. **Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review**. *ACM Trans. Softw. Eng. Methodol.*, 32, 2, Article 51, (March 2023), 61 pages. DOI: 10.1145/3542945.

[8] Janis Arents, Valters Abolins, Janis Judvaitis, Oskars Vismanis, Aly Oraby, and Kaspars Ozols. 2021. **Human-Robot Collaboration Trends and Safety Aspects: A Systematic Review**. *Journal of Sensor and Actuator Networks*, 10, 3, (July 2021). DOI: 10.3390/jsan10030048.

[9] Julia M. Badger, Dustin Gooding, Kody Ensley, Kimberly A. Hambuchen, and Allison Thackston. 2016. **ROS in Space: A Case Study on Robonaut 2**. In *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Springer, 343–373. DOI: 10.1007/978-3-319-26054-9_13.

[10] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage. 2006. **Performance Prediction of Component-Based Systems**. In *Architecting Systems with Trustworthy Components*. Springer, 169–192. DOI: 10.1007/11786160_10.

[11] Steffen Becker, Heiko Koziolek, and Ralf Reussner. 2009. **The Palladio component model for model-driven performance prediction**. *Journal of Systems and Software*, 82, 1, 3–22. Special Issue: Software Performance - Modeling and Analysis. DOI: 10.1016/j.jss.2008.03.066.

[12] L.A. Belady and C.J. Evangelisti. 1981. **System partitioning and its measure**. *Journal of Systems and Software (JSS)*, 2, 1, 23–29. DOI: 10.1016/0164-1212(81)90043-1.

[13] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. 2011. **Safety, dependability and performance analysis of extended aadl models**. *The Computer Journal*, 54, 5, (May 2011), 754–775. DOI: 10.1093/comjnl/bxq024.

[14] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Andres Orebäck. 2005. **Towards component-based robotics**. In *International Conference on Intelligent Robots and Systems (IROS '05)*. IEEE, 163–168. DOI: 10.1109/IROS.2005.1545523.

[15] Franz Brosch, Heiki Koziolek, Barbora Buhnova, and Ralf Reussner. 2012. **Architecture-Based Reliability Prediction with the Palladio Component Model**. *IEEE Transactions on Software Engineering (TSE)*, 38, 6, (November 2012), 1319–1339. DOI: 10.1109/TSE.2011.94.

[16] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. 2011. **Automated Extraction of Architecture-LevelPerformance Models of DistributedComponent-Based Systems**. In *International Conference on Automated Software Engineering* (ASE '11). IEEE, 183–192. DOI: 10.1109/ASE.2011.6100052.

[17] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. 2009. **Automated Extraction of Palladio Component Models from Running Enterprise Java Applications**. In *International Conference on Performance Evaluation Methodologies and Tools* (VALUETOOLS '09) Article 10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 10 pages. DOI: 10.4108/ICST.VALUETOOLS2009.7981.

[18] Davide Brugali. 2015. **Model-Driven Software Engineering in Robotics**. *IEEE Robotics & Automation Magazine*, 22, 3, 155–166. DOI: 10.1109/MRA.2015.2452201.

[19] Davide Brugali, Alex Brooks, Anthony Cowley, Carle Côté, Antonio C. Domínguez-Brito, Dominic Létourneau, Françis Michaud, and Christian Schlegel. 2007. **Trends in Component-Based Robotics**. In *Software Engineering for Experimental Robotics*. Springer, 135–142. DOI: 10.1007/978-3-540-68951-5_8.

[20] Paulo Canelas, Miguel Tavares, Ricardo Cordeiro, Alcides Fonseca, and Christopher S. Timperley. 2022. **An Experience Report on Challenges in Learning the Robot Operating System**. In *International Workshop on Robotics Software Engineering (RoSE '22)*, 33–38. DOI: 10.1145/3526071.3527521.

[21] Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. 2008. **Reverse Engineering Software-Models of Component-Based Systems**. In *European Conference on Software Maintenance and Reengineering (CSMR '08)*. IEEE, 93–102. DOI: 10.1109/CSMR.2008.4493304.

[22] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Judith Stafford, Reed Little, and Robert Nord. 2003. **Documenting Software Architectures: Views and Beyond**. Addison-Wesley Professional.

[23] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. 2011. **Investigating the use of lexical information for software system clustering**. In *European Conference on Software Maintenance and Reengineering (CSMR '11)*. IEEE, 35–44. DOI: 10.1109/CSMR.2011.8.

[24] Martin Dahl, Kristofer Bengtsson, Martin Fabian, and Petter Falkman. 2017. **Automatic Modeling and Simulation of Robot Program Behavior in Integrated Virtual Preparation and Commissioning**. *Procedia Manufacturing*, 11, 284–291. International Conference on Flexible Automation and Intelligent Manufacturing (FAIM '17). DOI: 10.1016/j.promfg.2017.07.107.

[25] Edson de Araújo Silva, Eduardo Valentin, Jose Reginaldo Hughes Carvalho, and Raimundo da Silva Barreto. 2021. **A survey of Model Driven Engineering in robotics**. *Journal of Computer Languages*, 62, 101021. DOI: 10.1016/j.cola.2020.101021.

[26] D. Doval, S. Mancoridis, and B.S. Mitchell. 1999. **Automatic clustering of software systems using a genetic algorithm**. In *International Workshop on Software Technology and Engineering Practice (STEP '99)*. IEEE, 73–81. DOI: 10.1109/STEP.1999.798481.

[27] Pablo Estefo, Jocelyn Simmonds, Romain Robbes, and Johan Fabry. 2019. **The Robot Operating System: package reuse and community dynamics**. *Journal of Systems and Software (JSS)*, 151, 226–242. DOI: 10.1016/j.jss.2019.02.024.

[28] Roy S. Freedman. 1991. **Testability of software components**. *IEEE Trans. Softw. Eng.*, 17, 6, (June 1991), 553–564. DOI: 10.1109/32.87281.

[29] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. 2011. **Enhancing architectural recovery using concerns**. In *International Conference on Automated Software Engineering (ASE '11)*. IEEE, 552–555. DOI: 10.1109/ASE.2011.6100123.

[30] Xiaocheng Ge, Richard F. Paige, and John A. McDermid. 2010. **Analysing System Failure Behaviours with PRISM**. In *International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI '10)*, 130–136. DOI: 10.1109/SSIRI-C.2010.32.

[31] Negar Ghorbani, Joshua Garcia, and Sam Malek. 2019. **Detection and repair of architectural inconsistencies in java**. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 560–571. DOI: 10.1109/ICSE.2019.00067.

[32] Brian Goldfain, Paul Drews, Changxi You, Matthew Barulic, Orlin Velev, Panagiotis Tsiotras, and James M Rehg. 2019. **AutoRally: An Open Platform for Aggressive Autonomous Driving**. *IEEE Control Systems Magazine*, 39, 1, 26–55. DOI: 10.1109/MCS.2018.2876958.

[33] Adriano Gomes, Alexandre Mota, Augusto Sampaio, Felipe Ferri, and Julio Buzzi. 2010. **Systematic model-based safety assessment via probabilistic model checking**. In *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer, 625–639.

[34] Raju Halder, José Proença, Nuno Macedo, and André Santos. 2017. **Formal Verification of ROS-Based Robotic Applications Using Timed-Automata**. In *International FME Workshop on Formal Methods in Software Engineering (FormaliSE '17)*, 44–50. DOI: 10.1109/FormaliSE.2017.9.

[35] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. 1995. **Reverse Engineering to the Architectural Level**. In *International Conference on Software Engineering (ICSE '95)*. IEEE, 186–186. DOI: 10.1145/225014.225032.

[36] Abdelfetah Hentout, Mustapha Aouache, Abderraouf Maoudj, and Isma Akli. 2019. **Human-robot interaction in industrial collaborative robotics: a literature review of the decade 2008–2017**. *Advanced Robotics*, 33, 15-16, 764–799. DOI: 10.1080/01691864.2019.1636714.

[37] D.H. Hutchens and V.R. Basili. 1985. **System Structure Analysis: Clustering with Data Bindings**. *Transactions on Software Engineering (TSE)*, SE-11, 8, 749–757. DOI: 10.1109/TSE.1985.232524.

[38] Casidhe Hutchison, Milda Zizyte, Patrick E. Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. 2018. **Robustness Testing of Autonomy Software**. In *International Conference on Software Engineering: Software Engineering in Practice* (ICSE-SEIP '18). ACM, 276–285. DOI: 10.1145/3183519.3183534.

[39] Felix Ingrand. 2019. **Recent Trends in Formal Validation and Verification of Autonomous Robots Software**. In *International Conference on Robotic Computing (IRC)*, 321–328. DOI: 10.1109/IRC.2019.00059.

[40] Tauseef Israr, Murray Woodside, and Greg Franks. 2007. **Interaction tree algorithms to extract effective architecture and layered performance models from traces**. *Journal of Systems and Software*, 80, 4, 474–492. Software Performance. DOI: 10.1016/j.jss.2006.07.019.

[41] Bernard C. Jiang and Charles A. Gainer. 1987. **A Cause-and-Effect Analysis of Robot Accidents**. *Journal of Occupational Accidents*, 9, 1, 27–45. DOI: 10.1016/0376-6349(87)90023-X.

[42] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian Elbaum, and Yonghwi Kwon. 2021. **Swarmbug: Debugging Configuration Bugs in Swarm Robotics**. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. ACM, 868–880. DOI: 10.1145/3468264.3468601.

[43] Jin Hwa Jung and Dong-Geon Lim. 2020. **Industrial robots, employment growth, and labor cost: a simultaneous equation analysis**. *Technological Forecasting and Social Change*, 159, 120202. DOI: 10.1016/j.techfore.2020.120202.

[44] Min Yang Jung, Anton Deguet, and Peter Kazanzides. 2010. **A component-based architecture for flexible integration of robotic systems**. In *International Conference on Intelligent Robots and Systems (IROS '10*, 6107–6112. DOI: 10.1109/IROS.2010.5652394.

[45] Sayali Kate, Michael Chinn, Hongjun Choi, Xiangyu Zhang, and Sebastian Elbaum. 2021. **PHYSFRAME: Type Checking Physical Frames of Reference for Robotic Systems**. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. ACM, 45–56. DOI: 10.1145/3468264.3468608.

[46] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian Elbaum, and Zhaogui Xu. 2018. **Phys: Probabilistic Physical Unit Assignment and Inconsistency Detection**. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. ACM, 563–573. DOI: 10.1145/3236024.3236035.

[47] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. 2015. **An Open Approach to Autonomous Vehicles**. *IEEE Micro*, 35, 6, 60–68. DOI: 10.1109/MM.2015.133.

[48] Mourad Kmimech, Mohamed Tahar Bhiri, and Phillipe Aniorte. 2009. **Checking Component Assembly in Acme: An Approach Applied on UML 2.0 Components Model**. In *International Conference on Software Engineering Advances (ICSEA '09)*, 494–499. DOI: 10.1109/ICSEA.2009.78.

[49] Sophia Kolak, Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher S Timperley. 2020. **It Takes a Village to Build a Robot: An Empirical Study of The ROS Ecosystem**. In *International Conference on Software Maintenance and Evolution (ICSME '20)*. IEEE, 430–440. DOI: 10.1109/ICSME46990.2020.00048.

[50] Philip Koopman and Michael Wagner. 2016. **Challenges in autonomous vehicle testing and validation**. *SAE International Journal of Transportation Safety*, 4, 1, 15–24. http://www.jstor.org/stable/26167741.

[51] Heiko Koziolek. 2010. **Performance evaluation of component-based software systems: A survey**. *Performance Evaluation*, 67, 8, 634–658. Special Issue on Software and Performance. DOI: 10.1016/j.peva.2009.07.007.

[52] James Kramer and Matthias Scheutz. 2007. **Development environments for autonomous mobile robots: A survey**. *Autonomous Robots*, 22, 2, 101–132. DOI: 10.1007/s10514-006-9013-8.

[53] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. **Automatic Mining of Specifications from Invocation Traces and Method Invariants**. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2014). ACM, 178–189. DOI: 10.1145/2635868.2635890.

[54] Marta Kwiatkowska, Gethin Norman, and David Parker. 2009. **PRISM: Probabilistic Model Checking for Performance and Reliability Analysis**. *SIGMETRICS Perform. Eval. Rev.*, 36, 4, (March 2009), 40–45. DOI: 10.1145/1530873.1530882.

[55] Marta Kwiatkowska, Gethin Norman, and David Parker. 2002. **Prism: probabilistic symbolic model checker**. In *Computer Performance Evaluation: Modelling Techniques and Tools*. Springer, 200–204.

[56] William Landi. 1992. **Undecidability of Static Analysis**. *ACM Lett. Program. Lang. Syst.*, 1, 4, (December 1992), 323–337. DOI: 10.1145/161494.161501.

[57] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. **General LTL Specification Mining (T)**. In *International Conference on Automated Software Engineering (ASE '15)*, 81–92. DOI: 10.1109/ASE.2015.71.

[58] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. 2019. **Formal Specification and Verification of Autonomous Robotic Systems: A Survey**. *ACM Comput. Surv.*, 52, 5, Article 100, (September 2019), 41 pages. DOI: 10.1145/3342355.

[59] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. **Robot Operating System 2: Design, Architecture, and Uses In The Wild**. *Science Robotics*, 7, 66, eabm6074. DOI: 10.1126/scirobotics.abm6074.

[60] Ivano Malavolta, Grace Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. 2020. **How Do You Architect Your Robots? State of the Practice and Guidelines for ROS-Based Systems**. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '20)*. ACM, 31–40. DOI: 10.1145/3377813.3381358.

[61] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn R. Chen, and Emden R. Gansner. 1999. **Bunch: a clustering tool for the recovery and maintenance of software system structures**. In *International Conference on Software Maintenance (ICSM '99)*. IEEE, 50–59. DOI: 10.1109/ICSM.1999.792498.

[62] Xinjun Mao, Hao Huang, and Shuo Wang. 2020. **Software Engineering for Autonomous Robot: Challenges, Progresses and Opportunities**. In *Asia-Pacific Software Engineering Conference (APSEC '20)*, 100–108. DOI: 10.1109/APSEC51365.2020.00018.

[63] Onaiza Maqbool and Haroon Babri. 2007. **Hierarchical Clustering for Software Architecture Recovery**. *Transactions on Software Engineering (TSE)*, 33, 11, 759–780. DOI: 10.1109/TSE.2007.70732.

[64] Onaiza Maqbool and Haroon Babri. 2004. **The weighted combined algorithm: a linkage algorithm for software clustering**. In *European Conference on Software Maintenance and Reengineering (CSMR '04)*. IEEE, 15–24. DOI: 10.1109/CSMR.2004.1281402.

[65] A. Marburger and D. Herzberg. 2001. **E-CARES research project: understanding complex legacy telecommunication systems**. In *European Conference on Software Maintenance and Reengineerin (CSMR '01)*. IEEE, 139–147. DOI: 10.1109/CSMR.2001.914978.

[66] Johan Moe and David A. Carr. 2001. **Understanding distributed systems via execution trace data**. In *International Workshop on Program Comprehension (IWPC '01)*. IEEE, 60–67. DOI: 10.1109/WPC.2001.921714.

[67] Chris Newcombe. 2014. **Why Amazon Chose TLA +** . In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 25–39. DOI: 10.1007/978-3-662-43652-3_3.

[68] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. **How Amazon Web Services Uses Formal Methods**. *Commun. ACM*, 58, 4, (March 2015), 66–73. DOI: 10.1145/2699417.

[69] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. 2014. **Behavioral Resource-Aware Model Inference**. In *International Conference on Automated Software Engineering* (ASE '14). ACM, 19–30. DOI: 10.1145/2642937.2642988.

[70] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2017. **Lightweight Detection of Physical Unit Inconsistencies without Program Annotations**. In *SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. ACM, 341–351. DOI: 10.1145/3092703.3092722.

[71] Samuel Parra, Sven Schneider, and Nico Hochgeschwender. 2021. **Specifying QoS Requirements and Capabilities for Component-Based Robot Software**. In *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE'21)*, 29–36. DOI: 10.1109/RoSE52553.2021.00012.

[72] Sukesh Patel, William Chu, and Rich Baxter. 1992. **A Measure for Composite Module Cohesion**. In *International Conference on Software Engineering (ICSE '92)*. ACM, 38–48. DOI: 10.1145/143062.143086.

[73] Morgan Quigley. 2009. **ROS: an open-source Robot Operating System**. In *International Conference on Robotics and Automation Workshop on Open Source Software*. http://lars.mec.ua.pt/public/LAR%20Projects/BinPicking/2016_RodrigoSalgueiro/LIB/ROS/icraoss09-ROS.pdf.

[74] André Santos, Alcino Cunha, and Nuno Macedo. 2019. **Static-Time Extraction and Analysis of the ROS Computation Graph**. In *International Conference on Robotic Computing (IRC '19)*. IEEE, 62–69. DOI: 10.1109/IRC.2019.00018.

[75] André Santos, Alcino Cunha, Nuno Macedo, Rafael Arrais, and Filipe Neves dos Santos. 2017. **Mining the usage patterns of ROS primitives**. In *International Conference on Intelligent Robots and Systems (IROS '17)*. IEEE, 3855–3860. DOI: 10.1109/IROS.2017.8206237.

[76] André Santos, Alcino Cunha, Nuno Macedo, and Cláudio Lourenço. 2016. **A framework for quality assessment of ROS repositories**. In *International Conference on Intelligent Robots and Systems (IROS '16)*. IEEE, 4491–4496. DOI: 10.1109/IROS.2016.7759661.

[77] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. 2006. **Discovering Architectures from Running Systems**. *Transactions on Software Engineering (TSE)*, 32, 7, (July 2006). DOI: 10.1109/TSE.2006.66.

[78] Robert W. Schwanke. 1991. **An Intelligent Tool for Re-Engineering Software Modularity**. In *International Conference on Software Engineering (ICSE '91)*. IEEE, 83–92. DOI: 10.1109/ICSE.1991.130626.

[79] Diksha Singh, Esha Trivedi, Yukti Sharma, and Vandana Niranjan. 2018. **TurtleBot: Design and Hardware Component Selection**. In *International Conference on Computing, Power and Communication Technologies (GUCON '18)*. IEEE, 805–809. DOI: 10.1109/GUCON.2018.8675050.

[80] Zipani Tom Sinkala and Sebastian Herold. 2021. **InMap: Automated Interactive Code-to-Architecture Mapping Recommendations**. In *International Conference on Software Architecture (ICSA '21)*. IEEE, 173–183. DOI: 10.1109/ICSA51549.2021.00024.

[81] Bridget Spitznagel and David Garlan. 1998. **Architecture-Based Performance Analysis**. In *Conference on Software Engineering and Knowledge Engineering* (SEKE '98). (June 1998). http://www.cs.cmu.edu/afs/cs/project/able/ftp/perform-seke98/perform-seke98.pdf.

[82] Christopher S. Timperley, **Tobias Dürschmid**, Bradley Schmerl, David Garlan, and Claire Le Goues. 2022. **ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems**. In *IEEE International Conference on Software Architecture (ICSA '22)*. IEEE, 112–123. DOI: 10.1109/ICSA53651.2022.00019.

[83] Christopher S. Timperley, Gijs van der Hoorn, André Santos, Harshavardhan Deshpande, and Andrzej Wasowski. [n. d.] **ROBUST: 221 Bugs in the Robot Operating System**.

[84] Milos Vasic and Aude Billard. 2013. **Safety Issues in Human-Robot Interactions**. In *International Conference on Robotics and Automation (ICRA '13)*. IEEE, 197–204. DOI: 10.1109/ICRA.2013.6630576.

[85] Valeria Villani, Fabio Pini, Francesco Leali, and Cristian Secchi. 2018. **Survey on Human-Robot Collaboration in Industrial Settings: Safety, Intuitive Interfaces and Applications**. *Mechatronics*, 55, 248–266. DOI: 10.1016/j.mechatronics.2018.02.009.

[86] Markus Weißmann, Stefan Bedenk, Christian Buckl, and Alois Knoll. 2011. **Model Checking Industrial Robot Systems**. In *Model Checking Software*. Springer, 161–176. DOI: 10.1007/978-3-642-22306-8_11.

[87] Melonee Wise, Michael Ferguson, Derek King, Eric Diehr, and David Dymesich. 2016. **Fetch & Freight: Standard Platforms for Service Robot Applications**. In *Workshop on autonomous mobile service robots*. http://docs.fetch3staging.wpengine.com/FetchAndFreight2016.pdf.

[88] Thomas Witte and Matthias Tichy. 2018. **Checking Consistency of Robot Software Architectures in ROS**. In *International Workshop on Robotics Software Engineering (RoSE '18)*. IEEE, 1–8. https://ieeexplore.ieee.org/document/8445812.