ROSInfer: Statically Inferring Behavioral Component Models for ROS-based Robotics Systems

Tobias Dürschmid, Christopher S. Timperley, David Garlan, and Claire Le Goues
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA, USA

ABSTRACT

Robotics systems are complex, safety-critical systems that can consist of hundreds of software components that interact with each other dynamically during run time. Software components of robotics systems often exhibit reactive, periodic, and state-dependent behavior. Incorrect component composition can lead to unexpected behavior, such as components passively waiting for initiation messages that never arrive. Model-based software analysis is a common technique to identify incorrect behavioral composition by checking desired properties of given behavioral models that are based on component state machines. However, writing state machine models for hundreds of software components manually is a labor-intensive process. This motivates work on automated model inference. In this paper, we present an approach to infer behavioral models for systems based on the Robot Operating System (ROS) using static analysis by exploiting assumptions about the usage of the ROS API and ecosystem. Our approach is based on searching for common behavioral patterns that ROS developers use for implementing reactive, periodic, and state-dependent behavior using the ROS framework API. We evaluate our approach and our tool `ROSInfer` on five complex real-world ROS systems with a total of 534 components. For this purpose we manually created 155 models of components from the source code to be used as a ground truth and available data set for other researchers. `ROSInfer` can infer causal triggers for 87 % of component architectural behaviors in the 534 components.

ACM Reference Format:
Tobias Dürschmid, Christopher S. Timperley, David Garlan, and Claire Le Goues. 2024. `ROSInfer`: Statically Inferring Behavioral Component Models for ROS-based Robotics Systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639206

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 [30, 35, 52, 79] their faults can potentially cause physical damage, injuries, and even fatalities [5, 49, 78]. However, ensuring that robotics software systems are safe and operate correctly is hindered by their large size and complexity [42, 54].

Robotics systems, especially systems written for the Robot Operating System (ROS) [66], 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*, i.e., asynchronous data exchange [2, 13, 36, 66, 76]. Robotics systems can be comprised of hundreds of software components, each of which can have complex behavior [10, 45, 76]. Many ROS systems are predominantly composed of reusable components implemented by external developers [41]. In this context, the main challenge is their correct composition [14, 76].

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 [14, 27, 75]. In this paper we call these bugs "*behavioral architectural composition bugs*", because they are caused by inconsistent compositions and manifest in inter-component communication, i.e., on the architectural level. Finding and debugging behavioral architectural composition bugs in robotics systems is challenging, because components frequently fail silently, failures can propagate through the system, and state-dependent behavior is hard to reason about [1, 17, 31, 42].

Fortunately, a large amount of existing formal methods research has been done on using formal model-based architecture analysis to ensure the safety and correct composition of components [4, 12, 20, 32, 51, 59, 60, 80]. Based on structural and behavioral models, such as state machines, of the current system, architects can find inconsistencies or predict the impact of changes on the system's behavior using existing analyses [9, 11, 23, 26, 27, 40, 47, 83]. However, in practice, due to the complexity of robotics systems, creating models manually and keeping them consistent with the code is time-consuming and difficult [19, 20, 80].

To reduce the modeling effort and make formal analysis more accessible in practice, this paper presents a static analysis technique to infer architecturally-relevant behavior for ROS-based systems. *Architecturally-relevant component behavior* is the set of all behaviors required to describe what causes inter-component communication, i.e., what causes a component to send messages.

Architectural recovery techniques, such as ROSDiscover [76], HAROS [67, 69], and the tool by Witte et al. [82], can reconstruct structural models, such as component-port-connector models that describe the organization of components and the relationships between them, including what types of inputs and outputs the component handles. However, they do not reconstruct *component behavior*, i.e., dynamic aspects that describe how the component

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639206

reacts to inputs and how it produces 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 [61], 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. Furthermore, for complex systems they come with the trade-off of long analysis times or low coverage.

To address the challenge of automatically inferring behavioral component models for ROS-based systems we propose to use static analysis of the system's source code written in C++. Static inference of behavioral models is challenging, because the analysis needs to infer what subset of a component's code communicates with other components, i.e., is *architecturally-relevant*, and what causes message sending behavior, i.e., in what situations is the code that implements inter-component communication executed.

Fortunately, the following observations about the ROS ecosystem make ROS components easier to analyze than general C++ code:

- Inter-component communication happens almost exclusively via Application Programming Interface (API) calls that have well-understood architectural semantics [68].
- The causes of message sending behavior (e.g., periodic loops, reacting to receiving a message) are usually implemented using features provided by the ROS framework and often follow common behavioral patterns.

Based on these observations, we make the following contributions:

(1) An approach of API-call-guided static analysis to infer state machine models that capture architecturally-relevant component behavior for ROS systems (in Section 3).
(2) A prototypical implementation of the approach that is an extension to ROSDiscover, an existing tool for structural architecture recovery [76] (in replication package).
(3) An empirical evaluation of the presented approach's recovery rate, recall, and precision of five real-world open-source systems with a total of 534 components (in Section 4).
(4) A data set of 155 handwritten behavioral component models of components for the evaluation systems that can be used by other researchers (in replication package).

While this work focuses on the ROS ecosystem, the approach of API-call guided static recovery of component behavioral models seems promising to generalize to other frameworks and ecosystems that follow the observations above.

2 ARCHITECTURALLY-RELEVANT COMPONENT BEHAVIOR IN ROS

Fortunately, only a small part of the overall behavior of a component is relevant to describe to component's behavior on an architectural level. This makes it practically possible for static analysis to infer behavioral component models for complex systems. This section defines architecturally-relevant behavior and describes corresponding architectural styles offered by ROS.

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).

2.1 The Robot Operating System (ROS)

ROS is the most popular framework for robotics systems, and supports a large ecosystem of more than 7 500[1] software packages. In practice, ROS is used by many companies, such as Amazon, BMW, Boeing, Bosch, Boston Dynamics, NASA, PAL Robotis, and Siemens.

From a software engineering research perspective, ROS is a typical example of a framework for component-based architectures. That means ROS systems are developed to consist of independently deployable run-time units i.e., *components*, in ROS known as *nodes*, that primarily communicate with each other via messages [2, 13, 36, 66, 76]. Each node is implemented as an independent process and is typically responsible for providing a single function (e.g., transforming depth images into point clouds, planning the robot's trajectory, and translating movements into low-level motor commands). Nodes communicate with each other over named channels (i.e., topics, services, actions). In this paper, we focus on topic-based communications and service calls, which represent the vast majority of communications in ROS systems [76].

Services represent a synchronous two-way call-return-style.

Topics use a publish-subscribe model to provide asynchronous message-based, multi-endpoint communication between nodes. Each topic can have multiple publishers and subscribers. Node-topic connections are defined in the node's source code and are established at run time by providing the name of a topic in the form of a string to the ROS API. Topics are typically used for both reporting periodic information (e.g., camera data, sensor data, position) and sporadic requests (e.g., disabling a motor).

Figure 1 shows a typical example of how ROS developers use the ROS API to implement architectural component behavior. To define behavior that handles input messages, the ROS framework lets users register callback methods that are called by the framework when a component receives a message (see `subscribe` call in Figure 1). To define periodic behavior, ROS offers the Timer API and the Rate API (see periodic sleep in Figure 1).

2.2 Formalization of Architecturally-Relevant Component Behavior

To define the semantics of the behavioral models that we infer, this section introduces the formalism of architecturally-relevant component behavior used throughout the paper. To tailor it to the domain of component-based robotics systems, we model architecturally-relevant behavior as a variant of input-output state machines that contain explicit state variables and include periodic triggers and component-start events with the following definitions:

[1] https://index.ros.org/stats/ [Date Accessed: 27th July 2023]

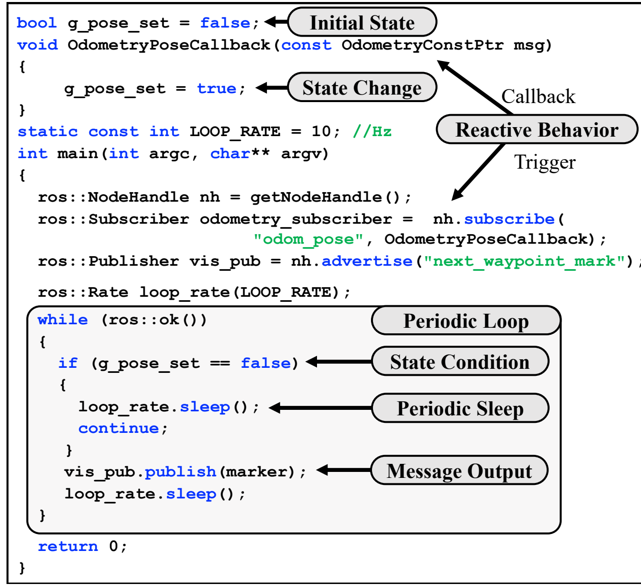

Figure 1: Simplified example of a ROS component (lattice_trajectory_gen) that waits for an input message and then periodically publishes a message with a frequency of 10 Hz.

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, we do not model the content of messages.

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.

$$S = [g_pose_set : Bool]; s_0 = [g_pose_set = false]$$
$$M_{in} = \{odom_pose\}; P = \{p_{10}\}$$
$$M_{out} = \{next_waypoint_mark\};$$
$$I = \{p_{10}, odom_pose\}; O = \{next_waypoint_mark, \epsilon\}$$

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 $o = next_waypoint_mark$

Figure 2: Example model for code shown in Figure 1. The first transition handles *odom_pose* inputs and changes the state variable g_pose_set to true without an output. The second transition triggers periodically with a frequency of 10 Hz if the state variable g_pose_set is true. Then it sends a message.

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: $\forall T \in Types : \top \in T$.

Figure 2 shows the model for the example code shown in Figure 1.

2.3 Behavioral Architecture Composition Bugs

Behavioral component models can be used to find *behavioral architecture composition bugs*, i.e., bugs that result from incorrect component composition and impact the software architecture's behavior. Figure 1 shows an example of a bug from the Autoware.AI [39] 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. Approaches that recover only component-port-connector (CPC) models, such as ROSDiscover [76], cannot find this bug, because they cannot infer that the input is *required*, i.e., the component's main functionality depends on it. Our approach classifies the input as required by inferring component state machines; identifying that the component's expected behavior happens in the state `g_pose_set == true`, which is triggered only by the message input; and detecting that no other component sends this message.

Compared to ROSDiscover, `ROSInfer` adds models of internal component behaviors in the format of component state machines supporting the finding of behavioral architecture composition bugs. In particular, it can find three real-world bugs from the original ROSDiscover evaluation, that ROSDiscover was unable to find (autoware-02, autoware-03, and autoware-10) [76].

Besides these Autoware bugs, our approach increases the practicality and accessibility of a large amount of existing analyses on component state machine models to find behavioral architecture

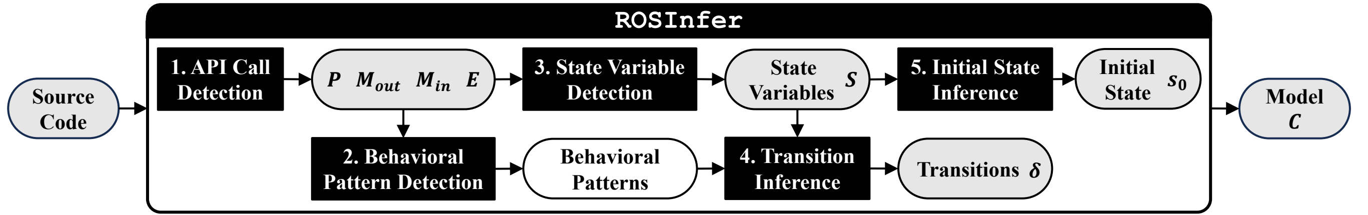

Figure 3: Overview of the ROSInfer static analysis. Boxes represent analysis steps (see Section 3), gray ovals represent elements of the output model (see Section 2.2), white ovals represent intermediate results, and arrows indicate data flow.

composition bugs [6, 7, 9, 23, 26, 27, 43, 44, 47, 74, 83], since roboticists do not need to create the required models manually. Thereby, it becomes easier for them to ensure the safety and correct composition of software components [2, 4, 12, 32, 51, 59, 60, 80].

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 [11] or logical reasoning [40].

Furthermore, synchronizing the resulting component state machines on their input/output messages creates a system-wide state machine model over which system-level Linear Temporal Logic (LTL) properties can be checked via approaches such as the ROS theorem prover by Kortik and Shastha [43] or PlusCal/TLA+ [48]. On these models, properties, such as a component changing to a desired state, no messages getting lost or ignored, or a component eventually publishing a certain message, can be checked [23, 26].

Additionally, knowing the frequencies at which periodic messages get published allows reasoning about the frequencies of transitive receivers of a message to check for unexpectedly high publishing frequencies (e.g., if behavior is not supposed to be periodic but reacts to periodically sent input).

3 APPROACH

This section describes our approach of API-call-guided static inference of architecturally-relevant component behavior for ROS systems and the implementation of this approach in our tool called `ROSInfer`. The analysis process is shown in Figure 3.

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 how the ROS framework is used in practice allow us to narrow down the analysis:

Component Framework API: Inter-component communication for sending and receiving messages is implemented almost exclusively via API calls that have well-understood architectural semantics [68, 76]. This simplifies the static inference of architecturally-relevant behavior by reducing the problem of tracing message-sending behavior to code locations to finding the corresponding API calls and inferring their arguments.

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 remainder of a periodic interval as shown in Figure 1). This simplifies the problem of identifying the triggers and effects of message sending / receiving behavior by looking for common patterns.

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 focuses 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.

3.1 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). `ROSInfer` detects API calls via AST matchers that look for method calls based on the fully qualified method name and argument list. For most kinds of API calls, `ROSInfer` then attempts to recover the values of arguments and the object on which the function is called to infer additional details, such as what port owns this behavior, or the frequency / duration of 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. For publish-subscribe styles, this consists of API calls to publish a message and corresponding API wrappers (e.g., `sendTransform`, `diagnostic_updater`, and `CameraPublisher`).

To identify the corresponding output port, `ROSInfer` infers the publisher object on which each publish call is made and traces the object to its creation via the `NodeHandle::advertise` API call.

Inferring Reactive Triggers M_{in}: To infer reactive triggers, behavioral inference needs 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 of detected input ports `ROSInfer` looks for callbacks that are parameters to the ROS API calls `subscribe`, `registerCallback`,[2] and `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 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++ 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 1), 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.

3.2 Behavioral Pattern Detection

After API call detection, `ROSInfer` runs a control-flow analysis on the program to construct 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 or a component event. To detect message outputs $out \in M_{out}$ reacting to message inputs $in \in M_{in}$ `ROSInfer` looks for the behavioral pattern of publish calls happening (transitively) within a subscriber callback method by checking for a path in the call graph from the callback method for *in* to any publish calls of *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. This pattern is shown in Figure 4.

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

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

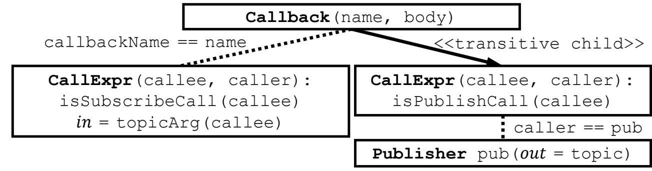

Figure 4: Simplified behavioral pattern to look for reactive message behavior.

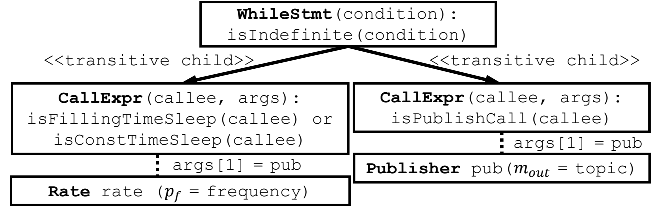

Figure 5: Simplified behavioral pattern to look for periodic behavior.

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, as these sleep calls happen periodically throughout the normal execution of the program. To identify unbounded loops `ROSInfer` considers loop conditions that are either `true` or `ros::ok()`. This pattern is shown in Figure 5.

3.3 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 1. 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 uses constant folding to replace variables and constants with the literals that they represent.

3.4 Transition Inference

After detecting state variables 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 constructs the union of all outputs and state changes with the same triggers and conditions.

3.5 Initial Value Inference

To infer the initial state $s_o \in S$ (i.e., the initial values for each state variable) of a 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 $\top$.

3.6 Implementation

We implemented ROSInfer using Clang for C/C++ as an extension of ROSDiscover [76]. (ROSDiscover and ROSInfer do not support Python, because C++ is the most used language in ROS [68].)

The ROS system is analyzed within a Docker container that contains the source code, ROS packages, and other dependencies. ROSInfer infers the API calls and its parameters, as well as AST elements such as while statements, if statements, and assignments and creates an abstraction of the component. Then it looks for the patterns described in the previous section and creates a JSON output that contains reactive, periodic, and state-based behavior from which the state machines described in Section 2.2 can be reconstructed. Finally, ROSInfer generates a PlusCal/TLA+ [48] specification containing the behavior and connectors of all components in the system configuration. To detect bugs, users can either specify their own LTL properties or provide a list of expected outputs to happen, from which ROSInfer auto-generates the corresponding LTL properties. The implementation is openly accessible on GitHub: https://github.com/cmu-rss-lab/rosdiscover-evaluation

4 EVALUATION

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.

4.1 Experimental Setup

To evaluate ROSInfer we asked the following research questions:

RQ 1 (Recovery Rate) — **Results in Section 4.2**

How high is ROSInfer's recovery rate for real-world ROS systems, i.e., what percentage of inferred architecturally-relevant behaviors 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 4.3**

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

As noted earlier, 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 4.4**

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 these research questions we evaluated ROSInfer on the data set presented in Timperley et al. [76], consisting of five large real-world open source systems:

Table 1: Systems used for evaluation with their stars on GitHub (as of 17th July 2023), lines of XML configuration files, lines of code including their dependent ROS packages, and number of components.

System	Stars	Lines of XML	Lines of Code	Components
AutoRally	685	43 455	190 340	26
Autoware	7100	30 771	250 509	230
Fetch	166	149 664	434 022	94
Husky	373	54 699	876 405	111
TurtleBot	279	1 237 887	1 596 546	108

Autware.AI [39], AutoRally [25], Fetch [81], Husky, and Turtlebot [72]. To demonstrate the complexity and size of the systems used for evaluation, statistics are shown in Table 1. Some components are part of multiple of these systems due to component reuse, leaving 550 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 Section 4.1. Due to the large size and complexity of these systems, we could not construct models for all 550 components. Therefore, we randomly picked components for each system (excluding components that are test or demo components that do not contain architecturally-relevant behavior). The first two authors of the paper (one who is closely familiar with the implementation of `ROSInfer` and one who is not) 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, special cases to look for, 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, the two authors who created the models measured their agreement of an overlap of 21 models (14.1 % of total) that were inferred by both authors, intentionally including some of the most complex models in this overlap. They agreed completely on 86 % of these components and partially on the remaining three components. After a discussion of the few differences in inferred models, they 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, an updated protocol for inferring models, and updating existing models to ensure their correctness.

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 threat by selecting diverse case studies with Autoware and AutoRally being mostly self-contained industrially-developed systems and Husky, Fetch, and Turtlebot being representative for typical open-source systems that are developed without central organization.

4.2 Measuring Recovery Rate (RQ1)

Methodology: As discussed in Section 3, `ROSInfer` denotes values that cannot be statically recovered with ⊤ to indicate unknown values. So the main metric for the recovery rate is how often ⊤ is included in parts of the resulting model.

We ran `ROSInfer` on all 550 components of the five systems presented in Section 4.1. Components that are included in multiple systems only count once. For 16 components the static analysis crashed due to errors in Clang, so these components are excluded from the evaluation, leaving 534. For 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 within uncalled callbacks (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 2**

In a exhaustive analysis of five large real-world ROS systems with 534 components **the overall trigger types recovery rate is 87 %**. The proportion of inferable values is 91 % for periodic rates, 100 % for reactive triggers, 72 % for state variable initial values, and 84 % for state changes.

Results: Detailed quantitative results are shown in Table 2. The high trigger types recovery rate confirms the Behavioral Pattern

Table 2: 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 534 components of the five large real-world systems presented in Section 4.1. 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 only once, even if they are included in multiple systems.

System	Trigger Types	Periodic Rates	Reactive Triggers	Initial States	State Changes
AutoRally	66.0 % (N = 47)	83.3 % (N = 18)	100.0 % (N = 13)	50.0 % (N = 4)	50.0 % (N = 2)
Autoware	89.0 % (N = 465)	92.9 % (N = 99)	100.0 % (N = 315)	65.4 % (N = 81)	80.9 % (N = 230)
Fetch	81.5 % (N = 27)	0.0 % (N = 1)	100.0 % (N = 21)	100.0 % (N = 3)	100.0 % (N = 2)
Husky	91.7 % (N = 48)	93.3 % (N = 15)	100.0 % (N = 29)	50.0 % (N = 8)	100.0 % (N = 6)
Turtlebot	84.2 % (N = 38)	66.7 % (N = 12)	100.0 % (N = 20)	80.0 % (N = 25)	100.0 % (N = 17)
All	**86.8 %** (N = 638)	**91.1 %** (N = 158)	**100.0 %** (N = 396)	**72.0 %** (N = 125)	**83.8 %** (N = 277)

Table 3: Recall and precision of `ROSInfer` based a comparison with 149 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	Ground Truth Component Models	Periodic Behaviors			Reactive Behaviors			State Variables			State Transitions		
		TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP
AutoRally	14	14	1	0	11	10	0	1	2	3	1	3	0
Autoware	119	22	2	0	146	19	15	30	10	8	42	13	6
Fetch	11	1	0	0	8	5	0	3	0	0	2	1	0
Turtlebot	5	2	0	0	2	3	2	0	2	0	0	3	0
All	149	39	3	0	167	37	17	34	14	11	45	20	6
Recall		**92.9 %** (of 42)			**81.9 %** (of 204)			**70.8 %** (of 48)			**69.2 %** (of 65)		
Precision		**100.0 %** (of 39)			**90.8 %** (of 184)			**75.6 %** (of 45)			**88.2 %** (of 51)		

Usage observation, because 87 % of all message sending behavior in the evaluated components could be automatically classified to conform to one behavioral pattern. 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 ot state variables 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.

4.3 Measuring Recall (RQ2)

Methodology: After creating the handwritten models as ground truth (see Section 4.1) 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 3

In a ground-truth comparison with 149 components `ROSInfer` has a recall of 93 % for periodic behavior, 82 % for reactive behavior, 71 % for state variables, and 69 % for state transitions.

Results: Detailed quantitative results are shown in Table 3.

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 6 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 is 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

```
lane_planner::vmap::VectorMap all_vmap;
void cache_point(const vector_map::PointArray& msg)
{
    all_vmap.points = msg.data;   <- State Change to Non-Empty
    update_values();
}
void update_values()
{                                 State Condition
    if (all_vmap.points.empty() || all_vmap.lanes.empty()
     || all_vmap.nodes.empty())
      return;                     Message Output
    [...]
    lane_planner::vmap::publish_add_marker([...]);
}
```

Figure 6: 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.

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 image object cannot be inferred statically.

4.4 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 3**

In a ground-truth comparison with 149 components `ROSInfer` has an precision of 100 % for periodic behavior, 91 % for reactive behavior, 76 % for state variables, and 88 % for state transitions.

Results: Detailed quantitative results are shown in Table 3.

False positives for reactive behavior are caused by a limitation of our current implementation that ignores state conditions of periodic behavior in `main` and therefore treats it as reacting to the event “component-started”, 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.

4.5 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 2.6 GHz) with 256 GB RAM, the static analysis took on average 36.5 s per component. The fully automated analysis of the entire Autoware.AI system took 3.8 h and much shorter for the other systems (Autorally: 10.2 min, Fetch: 29.9 min, Husky: 29.2 min, Turtlebot: 37.7 min). This indicates 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 model inference supports automatic regeneration of models when the sources change. Changes to the code base require regeneration of only the components that are affected by the 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 after the initial execution.

The effort it took to create the 155 handwritten models of this evaluation was approximately 120 hours of manual labor.[3] In practice, the developer time saved would be lower than 120 hours, 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, and economical.

5 DISCUSSION

In this section we discuss how the advantages and limitations of the approach fit into a practical software engineering context.

5.1 Lessons Learned about ROS Components

When building the behavioral models for ROS components and inspecting the root cause for missed behaviors we noticed:

(1) Many components are designed to process input streams and publish processed outputs like a pipes and filters architecture. These components are stateless and usually produce a single output for each input that they receive.
(2) Components that maintain states are often components that start to publish periodically after receiving a set of input messages that are used to initialize the component, such as the example shown in Figure 1.
(3) Only a few components implement a complex state machine. Most explicit or implicit state variables are booleans and only few components have more than three state variables.
(4) While the state machines that model the behavior of the component might be less complex, developers sometimes use more complex language features to express them than would be necessary (see Figure 6). This makes the code more extensible and easier to read by human developers, but harder to analyze using static analysis.

[3]Models were inferred by authors of this paper who have advanced knowledge of C++, a background in software architecture and formal modeling, are knowledgeable in robotics and Autoware.AI but have not been involved in its development. Model inference times will vary based on the expertise in the domain and experience with the system. This number is only intended to provide an informal estimate of the effort.

5.2 Incomplete Models

As discussed in the approach, inferred models can be incomplete, due to limitations discussed in the evaluation. There are two types of incompleteness: known unknowns (i.e., the analysis can infer the type of behavior but cannot reconstruct all its required elements so that the resulting model contains the keyword ⊤ representing an unknown value) and unknown unknowns (i.e., the analysis does not detect an instance of architecturally-relevant behavior so that this behavior is entirely missing from the resulting model). Examples for known unknowns are frequencies of periodic publishing, topic names, initial values or other assignments of state variables, or values that state variables are compared to in conditions. They occur when other variables are referenced that cannot be constant-folded, when C++ language features are used that the static analysis implementation does not support yet, when values are read from external sources, such as run-time inputs or files, or when developers follow the behavioral pattern but use too dynamic language features for static analysis to be able to identify the values.

In practice, users of `ROSInfer` can more easily deal with known unknowns, because `ROSInfer` directly points them to the place in the code for which it was unable to reconstruct the value. Users can then figure out the values and replace the known unknowns in the model with accurate values. Since they only need to fill in the blanks for some values, this task is much easier and less time-consuming than building the entire model from scratch. In some cases, known unknonws can be reduced with more engineering effort to improve the static analysis, but cannot be fully eliminated. Having incomplete models would still be preferable to having no models, because even incomplete models allow finding behavioral architecture composition bugs that would not have been found otherwise.

Unknown unknowns are more limiting in practice, since it is much harder for users to identify that information is missing from the generated models. Unknown unknowns can be reduced by extending the list of behavioral patterns to look for or by adding the APIs of commonly used libraries, but cannot be fully eliminated.

5.3 Real-World Bugs Found

To demonstrate the real-world effectiveness of `ROSInfer`, we identified documented behavioral architecture composition bugs within the data set presented in Timperley et al. [76], the only open set of architecture composition bugs in ROS currently known to us. Three of these bugs (autoware-02, autoware-03, autoware-10) can be classified as *behavioral* architecture composition bugs. Autoware-02 is shown in Figure 1. The other two also result from required inputs for important component behavior not being connected to publishers. We ran `ROSInfer` on these systems, generated PlusCal/TLA+ specifications and checked that expected outputs happen eventually. `ROSInfer` found all of these bugs based on a given list of components in the system configuration, a desired output to check for, and configuration parameter assignments. The resulting models are available in the supplemental material.

5.4 Coding Style Guidelines

Unlike many open-source ROS systems, most industrially developed projects follow coding style guidelines that narrow down the expected kinds of behaviors by telling developers to implement certain types of code in a certain way. We expect the recall of our approach to benefit from this, because fewer cases of unnecessarily complex versions of simpler code would exist. This effect can become even stronger if coding styles related to specifying architecturally relevant behavior are established, as almost all cases in which our approach cannot correctly infer architecturally-relevant behavior, the corresponding code could be refactored towards more analyzable code. For example, coding style guidelines, such as "component states should be explicitly modeled as variables in the code" to avoid the limitation described in Figure 6 by replacing `empty()` calls with a state variable, "state variables should be initialized explicitly" to avoid unknown or ambiguous initial states, or "ROS connectors should be used where possible" to avoid over-use of project-specific APIs. Similarly to how testability became a goal of software design to reduce the effort of ensuring correctness via testing, analyzability of code could become a future design goal of ROS code to supporting the automatic inference of rich behavioral models for automated formal analysis.

6 RELATED WORK

To demonstrate the novelty of this work, we discuss other analyses that have been performed on robotics systems, approaches to recover static architectures, and dynamic analyses of behavioral models and explain how our work differs from them.

6.1 Analysis of Robot Systems

Static analysis and formal model-based analysis have been used to automatically find bugs in robot systems before [4, 32, 51, 63]. For example, the systems Phriky [62], Phys [38], and Physframe [37] use type checking to find inconsistencies in assignments based on physical units or 3D transformations in ROS code.

Furthermore, Swarmbug [34] 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.

6.2 Inference of Structural Architectures

Most approaches for static recovery of software architectures reconstruct structural views of software modules from the perspective of a developer [3, 15, 18, 21, 22, 24, 28, 53, 55, 56, 64, 71, 73]. Since they show the code before compiling it, the module view does not show the relationships of components during run time [16] and therefore cannot find behavioral bugs.

DeSARM [65], SAMEtech [33] are dynamic approaches to reconstruct component-port-connector (CPC) models. ROSDiscover [76], HAROS [67, 69], and the tool by Witte et al. [82] can statically reconstruct CPC models for robotics systems. CPC 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, CPC 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, CPC models cannot be used to analyze the data flow within a system.

While CPC models identify the communication channels that components use to interact with the rest of the system they do not describe the conditions under which communications actually occur. They 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. In contrast, ROSInfer builds upon ROSDiscover's implementation and output to produce *behavioral component models* that describe how a node reacts to incoming messages (e.g., publishing a message to a different topic; switching into a different state) and timing-based triggers (e.g., publishing a status message every 50 ms). Therefore, ROSInfer allows to find behavioral architectural composition bugs, such as the ones described in Section 5.3 that ROSInfer found.

6.3 Dynamic Inference of Behaviors

Behavioral models of components can be inferred using dynamic analysis by observing the component behavior of representative execution traces. For example, Kieker [29, 77], DiscoTect [70] and Perfume [61] construct state machines from event traces obtained by run time monitoring. Similar approaches also use method invariants [46], LTL property templates [50] to increase the effectiveness. Domain-specific approaches have been proposed for for CORBA systems [58] or telecommunication systems [57]. The main limitation of dynamic approaches that are based on mere observation is that they can only measure 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 caused by an event. Furthermore, dynamic approaches require either an accurate simulator or real robot hardware to produce reliable results and 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 compared to static analysis approaches, such as ROSInfer.

7 CONCLUSIONS

In this paper we have shown that looking for specific API calls of the ROS framework and commonly used behavioral implementation patterns enable the effective static inference of models that capture architecturally-relevant component behavior with high precision and recall. 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, potentially leading to robotics systems becoming safer and more robust. Due to its potential to integrate well into practical software development environments with continuous integration and potentially higher accuracy with established coding style guidelines, we believe that API-call-guided static inference can have a significant impact on practice. While this paper focused only on ROS-based systems, we believe that API-call-based inference of component behavior is a promising approach that could generalize to other frameworks within the domain of cyber-physical systems and inspire future work that applies this approach to other ecosystems.

8 FUTURE WORK

We envision this contribution to enable the following future work:

Automatic Generation of Documentation: Automated inference of behavior models can support the generation of documentation for components, especially for reusable components. In cases in which components need to receive a set of initialization inputs to function properly (such as the example from Figure 1), inference of architecturally-relevant component models can be used to demonstrate which inputs are required for which output.

Automated Program Repair: Since models that were inferred from source code have the advantage of retaining a mapping between source code locations and model elements a repair patch for the model could be translated back to code. This opens motivates future work on model-repair translations back to code.

Repository Mining: Automatic inference of component behavioral models enables large-scale empirical research on the development and evolution of component behavior and inter-component communication patterns in complex robotics systems.

Test Generation: Information about what input messages change the component to a state in which it executes different behavior can be helpful to systematically generate test cases to cover a larger portion of the component's behavior.

Combination of Static and Dynamic Analysis: The limitations shown in the results of RQ1 can be overcome with the use of dynamic analysis. Furthermore, static analysis cannot infer execution times of tasks, producing models that cannot be used for most kinds of performance analysis, bottleneck analysis, or analysis of race condition. Results from from static analysis can inform systematic test generation and instrumentation of code to specifically obtain information that is missing in statically inferred models to increase recovery rate and recall. Furthermore, testing for the existence of inferred behavior can reduce false positives.

Generalization to Other Frameworks: The approach of API-call-based static inference of component behavior is not inherently specific to the ROS framework. Other component frameworks, such as NASA's FPrime framework [8], that provide APIs for component interaction mechanisms could implement this approach as well.

ACKNOWLEDGMENTS

The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. This work has been funded in part by the NSF under award numbers CCF-1750116 and CNS-2148301.

REFERENCES

[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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. 2018. F Prime: An Open-Source Framework for Small-Scale Flight Software Systems. In *Small Satellite Conference* number Advanced Technologies II, 328. https://digitalcommons.usu.edu/smallsat/2018/all2018/328/.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Zack Coker, David Gray Widder, Claire Le Goues, Christopher Bogart, and Joshua Sunshine. 2019. A Qualitative Study on Framework Debugging. In *International Conference on Software Maintenance and Evolution (ICSME '19)*, 568–579. DOI: 10.1109/ICSME.2019.00091.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Raju Halder, José Proença, Nuno Macedo, and André Santos. 2017. Formal Verification of ROS-Based Robotic Applications Using Timed-Automata. (2017). DOI: 10.1109/FormaliSE.2017.9.

[28] 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.

[29] Wilhelm Hasselbring and André van Hoorn. 2020. Kieker: A monitoring framework for software engineering research. *Software Impacts*, 5, 100019. DOI: https://doi.org/10.1016/j.simpa.2020.100019.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] Sitar Kortik and Tejas Kumar Shastha. 2021. Formal verification of ros based systems using a linear logic theorem prover. In *International Conference on Robotics and Automation (ICRA)*, 9368–9374. DOI: 10.1109/ICRA48506.2021.9561191.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] Leslie Lamport. 2009. The PlusCal Algorithm Language. In *Theoretical Aspects of Computing*. Springer, 36–60. DOI: 10.1007/978-3-642-03466-4_2.

[49] Larry A. Layne. 2023. Robot-related fatalities at work in the United States, 1992–2017. *American Journal of Industrial Medicine*, 66, 6, 454–461. DOI: https://doi.org/10.1002/ajim.23470.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] Jason Porter, Daniel A. Menascé, and Hassan Gomaa. 2021. A decentralized approach for discovering runtime software architectural models of distributed software systems. *Information and Software Technology*, 131, 106476. DOI: 10.1016/j.infsof.2020.106476.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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)*, 805–809. DOI: 10.1109/GUCON.2018.8675050.

[73] 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.

[74] 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.

[75] Christopher Timperley and A Wąsowski. 2019. 188 ROS bugs later: Where do we go from here? *ROSCON'19*. DOI: 10.36288/ROSCon2019-900898.

[76] Christopher S. Timperley, Dürschmid, Tobias, 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.

[77] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *International Conference on Performance Engineering (CPE '12)* (ICPE '12). ACM, 247–248. DOI: 10.1145/2188286.2188326.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] Kangfeng Ye, Ana Cavalcanti, Simon Foster, Alvaro Miyazawa, and Jim Woodcock. 2022. Probabilistic modelling and verification using RoboChart and PRISM. *Software and Systems Modeling*, 21, 2, (April 2022), 667–716. DOI: 10.1007/s10270-021-00916-8.