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

Tobias Dürschmid, Christopher S. Timperley, David Garlan, and Claire Le Goues
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 contain reactive, periodic, and state-dependent behavior that, when composed incorrectly, 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. 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 framework. 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` as a case study on Autoware.AI, a complex real-world ROS system for self-driving cars. For this purpose we manually created 118 models of components from the source code to be used as a ground truth and available data set for other researchers.

ACM Reference Format:
Tobias Dürschmid, Christopher S. Timperley, David Garlan, and Claire Le Goues. 2023. `ROSInfer`: Statically Inferring Behavioral Component Models for ROS-based Robotics Systems. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [38, 54, 80, 31] their faults can potentially cause dramatic harm to people [6, 36, 79]. However, ensuring that robotics systems are safe and operate correctly is hindered by their large size and complexity [45, 56].

ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Robotics systems, especially systems written for the Robot Operating System (ROS) [68], 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 [2, 39, 15, 68, 77]. They can be comprised of hundreds of software components, each of which can have complex behavior [12, 47, 77]. Many ROS systems are predominantly composed of reusable component implementation created by external developers [44]. In this context, the main challenge is their correct composition [77, 16].

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 [28, 16, 78]. In this paper 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 [45, 34, 1].

Software architects commonly use formal model-based architecture analysis to ensure the safety and correct composition of components [21, 14, 62, 63, 81, 53, 5, 35]. 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.

However, in practice, due to the complexity of robotics systems, creating models manually is time-consuming and difficult [81, 20, 21]. Furthermore, keeping models up-to-date while the software is evolving is an additional challenge of manual model creation [20]. This motivates work on automated component models recovery to reduce the modeling effort and make formal analysis more accessible in practice. In this paper we present a static analysis technique to infer behavioral component in the format of state machines from the code of ROS-based systems. We argue that these models can be used to find behavioral architectural composition bugs.

Architectural recovery techniques, such as ROSDiscover [77], HAROS [72, 70], 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 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 [64], 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 static analysis of the system's source code written in C++. In general, inferring behavioral models statically is undecidable [51]. 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. However, the following observations about the ROS ecosystem make this problem tractable for most commonly developed systems in practice:

(1) Inter-component communication happens almost exclusively via Application Programming Interface (API) calls that have well-understood architectural semantics [71]. Therefore, the resulting problem is reduced to finding potential control flow and data flow between this limited set of API calls.
(2) 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 template.

Based on these observations, this paper makes the following contributions:

(1) An approach to statically infer behavioral component models for ROS systems.
(2) A prototypical implementation of the approach that is an extension to ROSDiscover, an existing tool for structural architecture recovery [77].
(3) An empirical evaluation of the presented approach's precision on 106 components of our case study system Autoware.AI, the most complex open-source ROS project.
(4) A data set of 118 handwritten behavioral component models of components in Autoware.AI that can be used by other researchers.

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 provide APIs for inter-component interaction and in which observations about common behavioral patterns can be made.

The remainder of the paper is structured as follows: Section 2 discussed the background on ROS and work on which we build directly to set our work into context. Then, Section 3 describes the formalism of the models that we infer, how we infer them, and considerations we made when implementing our approach in our tool `ROSInfer`. In Section 4 we describe how we evaluate the effectiveness of our approach and implementation. In Section 5 we discuss how `ROSInfer` would integrate into the practical context of software development. Section 6 differentiates our work from existing work. Section 7 concludes this paper with a broader outlook on what this contributions means for software engineering and mentions future work that is enabled by our work and potential improvements that would make it more effective.

2 BACKGROUND

In this section, we provide background on ROS and describe ROSDiscover, the tool upon which `ROSInfer` is built. Further, we also describe the context of existing model-based analyses to set this work in context.

2.1 Robot Operating System

At a high level, ROS-based systems are composed of components known as *nodes*. 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). In ROS1, nodes communicate with each other over named communication channels (i.e., topics, services, actions) to form a functioning ensemble. In this paper, we focus our attention on topic-based communications, which represent the vast majority of communications in ROS.

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).

2.2 ROSDiscover

ROSDiscover [77] uses a static analyzer based on Clang to statically recover run-time ROS architectures. As part of its implementation, ROSDiscover uses a Clang-based static analyzer to obtain *structural models* of individual ROS nodes from their C++ source code. Each model structural description of a given node in terms of its interface (i.e., topics, services, and actions). Note that while those models identify the communication channels that the node uses to interact with the rest of the system via the ROS interface, they do not describe the conditions under which communications actually occur. In contrast, `ROSInfer` builds upon ROSDiscover's implementation and output to produce *behavioral models* of individual ROS nodes: `ROSInfer`'s behavioral models 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). `ROSInfer`'s models capture both the launch-time configuration and run-time state of a node and describe the conditions under which messages are sent.

2.3 Model-based Analyses

Software architects commonly use model-based architecture analysis to ensure the safety and correct composition of software components [2, 14, 62, 63, 81, 53, 5, 35]. 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 behavioral views (e.g., state machines or activity diagrams), and component-connector views, and deployment views [18]. 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-port-connector models [46, 7, 8, 11, 76, 13, 43] and state machines [49, 27, 25, 3]. 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.

3 APPROACH

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 2. 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 receivedt 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[1]
   (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.

3.1 Formalization of Component Behavior Models

To define the semantics of the behavioral models that we infer, this section introduces the formalism of behavioral component that we will use throughout the paper. The models are a variant of input-output state machines that describe the externally visible behavior of a component (i.e., the use of its ports). A graphical visualization of the model elements is shown in Figure 1.

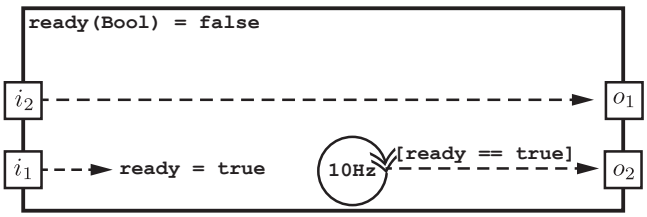

Figure 1: Conceptual Overview of the Component Model. Component outputs can be produced as a reaction to a component input or periodic trigger with given frequency. Both transitions can include conditions on typed component state variables that can be changed by transitions.

Each component state machine C is a 5-tuple $C = (S, s_0, I, O, T)$ of states S, an initial state $s_0 \in S$, input triggers I, outputs O, and transitions T.

The states S are combinations of typed state variables. Hence:

$$S = [var_1 : B_1, var_2 : B_2, ..., var_n : B_n]; \quad (1)$$

$$B_1, B_2, ..., B_n \in BasicTypes \quad (2)$$

$$BasicTypes = \{Bool, Int, Enum, Float, String\} \quad (3)$$

State variables correspond to non-constant code-level variables defined within the component that are accessible throughout the entire component implementation and that are required to define the operation of at least one transition in the state machine (i.e., are contained in at least one path condition of an output API call).

The input triggers I correspond to an event that triggers either a state change, or produces an output, or both. Input triggers are represented as either a message $m = (v \in T_p, p \in Port) \in M$ arriving at input port p with type $T_p \in PortTypes$ or a periodic trigger $p_f \in P$ with frequency $f \in Float$. Port types can be one of $BasicTypes$, or a compound type of those represented as a map $CompoundType \in String \rightarrow BasicTypes$.

$$PortTypes = BasicTypes \cup \{CompoundType\} \quad (4)$$

Hence:

$$I \subseteq M \cup P \quad (5)$$

Outputs $o = (v \in T_p, p \in Port)$ are represented as typed values v at output port p with type $T_p \in PortTypes$ or as the empty output ϵ for transitions that only change the state but don't produce an output.

$$O \subseteq M \cup \{\epsilon\} \quad (6)$$

Finally, the partial transition function $T \in S \times I \rightarrow O \times S$ is represented in the 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 following state $s' \in S$ in terms of s and i.

[1] 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.

```
bool ready = false;
void receive_initial(const Message msg)
{ // subscriber callback
  ready = true; // state transition
}
int main(int argc, char** argv)
{
  // port definitions
  ros::Subscriber sub = ros::Subscriber("t_sub", 10,
   receive_initial);
  ros::Publisher pub = ros::Publisher("t_pub", 10);

  const int local_LOOP_RATE = 10;
  ros::Rate loop_rate(local_LOOP_RATE);
  while (ros::ok())
  { // periodic loop
    if (!ready)
    { // state condition
      loop_rate.sleep();
      continue;
    }

    pub.publish(3); // message sending
    loop_rate.sleep();
  }
  return 0;
}
```

Figure 2: Simplified example of ROS code implementing a component that waits for an input message and then periodically publishes a message with a frequency of 10 Hz.

As an example, this is the resulting model of the code shown in Figure 2.

$$S = [ready : Bool] \tag{7}$$
$$s_0 = \{ready = false\} \tag{8}$$
$$Ports = \{in, out\} \tag{9}$$
$$M_{in} = Int \tag{10}$$
$$M_{out} = Int \tag{11}$$
$$I = \{p_{10}\} \cup in \times Int \tag{12}$$
$$O = out \times Int \tag{13}$$

The transitions ind pre- and post-condition form look like this:

receive_initial($s \in S, i \in I$):
pre: $i \in M_{in}$
post: $s' = s[ready = true]$ and $o = \epsilon$
This transition handles input at input port *in* and changes the state variable `ready` to true without an output.

periodic($s \in S, i \in I$):
pre: $i == p_{10} \wedge s[ready] == true$
post: $s' = s$ and $o = 3$
This transition triggers periodically with a frequency of 10 Hz if the state variable `ready` is true. Then it outputs the message "3".

Finally, the formalism needs a special element $\top$ (pronounced "top") that is used to represent an unknown value. In 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 will denote these unknown values with $\top$.

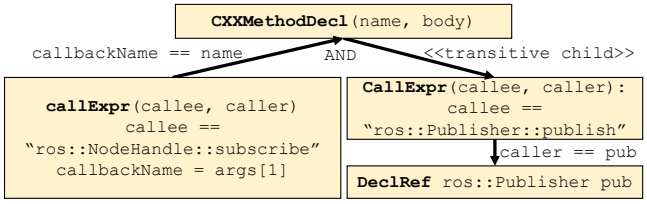

Figure 3: Simplified AST matching pattern to look for reactive triggers.

Section 3.2.1 describes how `ROSInfer` infers outputs O and transition post-conditions on O. Section 3.2.2 describes the inference of message inputs M and transition pre-conditions on M. Section 3.2.3 describes the inference of periodic inputs P and transition pre-conditions on P. Section 3.2.4 describes the inference of state variables S, the initial state s_0, and pre- and post-conditions on S.

3.2 Statically Inferring Component Behavior Model

3.2.1 Inferring Message Publishing. Architecturally-relevant behavior is the type of behavior that describes the outputs of a component. To statically infer architecturally relevant behavior, the first step is to identify what parts of the code are responsible for sending messages. In this work, we focus on publish-subscribe, the most common messaging style in ROS. To publish messages to a topic, the ROS API defines a `publish` call on the `Publisher` class. Every time this method is called the component sends a message to the publisher's topic. `ROSInfer` traverses the Abstract Syntax Tree (AST) of the component's source code to find all API calls to `ros::Publisher::publish` and all methods that transitively call the publish API call.

In terms of the formalism introduced in Section 3.1, this aspect of the approach infers the outputs O as well as the part of the transition post-conditions that produce outputs O.

3.2.2 Inferring Reactive Triggers. Reactive publishing behavior is message sending that is caused by receiving a message (note that receiving the message does not necessarily always have to cause message sending).

To infer reactive behavior, our analysis looks for control flow from subscriber callbacks to publish calls. Figure 3 shows a simplified pattern of the relationships between AST elements that `ROSInfer` is looking for. Subscriber callbacks define the component's behavior in response to receiving a certain message. In ROS the callback method names are arguments to the `subscribe` API call on the `Subscriber` class. `ROSInfer` first traverses the program's source code to find all calls to the `subscribe` API and retrieve the argument that corresponds to the method name of the callback. Then `ROSInfer` checks whether the subscriber callback method transitively calls publish. If so, the trigger for this message is reactive to receiving a message at the subscriber topic.

In terms of the formalism introduced in Section 3.1, this aspect of the approach infers the message inputs M as well as the part of the transition pre-conditions that depend on message inputs.

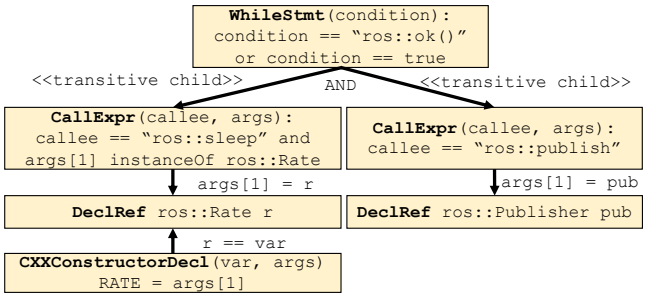

Figure 4: Simplified AST matching pattern to look for periodic triggers .

3.2.3 Inferring Periodic Triggers. Periodic publishing behavior is repeated sending of a message of the same type with a constant upper target frequency (note that message do not necessarily always have to be send every intervals can be prolonged).

To infer periodic behavior, our analysis looks for publish calls within loops that contain a sleep call. Figure 4 shows a simplified pattern of the relationships between AST elements that `ROSInfer` is looking for. The ROS API provides a `sleep` method for `Rate` objects. When this call is made the ROS framework suspends the execution of this thread for the remaining time of the interval that corresponds to the specified rate. So rather than waiting for a constant time, this API call makes sure that the execution does not happen more often than the specified target frequency. Our analysis looks for calls of this kind that happen within a loop that has a condition that is either `true` or `ros::ok()`. It then looks for publish calls that also happen within the body of this loop and marks those as being periodic.

Since frequency of the periodic behavior is specified in the constructor of the `Rate` object on which sleep is called, the analysis can infer the target frequency in all cases in which this value can be determined using constant-folding. If the values cannot be constant-folded, `ROSInfer` denotes these unknown values with $\top$.

In terms of the formalism introduced in Section 3.1, this aspect of the approach infers the periodic inputs P as well as the part of the transition pre-conditions that depend on periodic inputs.

3.2.4 Inferring State-dependent Behavior. A component's state is the set of pairs of run-time configuration parameters (i.e., *state variables*) and their values on which either periodic or reactive publishing behavior or changes to state variables depend. To express the conditions under which periodic and reactive behavior depends, an analysis needs to infer which state variables the component maintains, the conditions on these state variables that determine whether a message is sent or not, and the changes to state variables (i.e., *state transitions*).

For the purpose of static analysis, we use two heuristics to identify state variables as variables:

- **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. Hence the control conditions of all behaviors that are describing the post-condition of a transition define the pre-condition of those transitions. Therefore, only variables that are part of these pre-conditions can be state variables.
- **Scope Heuristic**: The variable is in global or compocanent-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. But this heuristic can lead to not detecting a state variable that is defined as a local variable of a function that defines the main loop of the component.

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.

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). The resulting compound condition is the pre-condition of the transition.

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. These can be either in their declarations or in the entry point of the component and its transitive calls. If an initial expression is found `ROSInfer` attempts to constant-fold the expression. Analogues to previous cases, values that cannot be constant-folded are denoted with $\top$.

In terms of the formalism introduced in Section 3.1, this aspect of the approach infers state variables S, their initial values resulting in s_0, as well as the part of the transition pre- and post-conditions that depend on or change state variables.

3.3 Implementation

We implemented `ROSInfer` using Clang for C/C++ as an extension of `ROSDiscover` [77]. The ROS system is analyzed within a Docker container that contains all 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 corresponds that contains reactive, periodic, and state-based behavior from which the state machines described in Section 3.1 can be reconstructed. The implementation is openly accessible on GitHub: Link blinded for review. Reviewers can find the implementation in the replication package.

3.4 Possible Analyses for Inferred Models

In the introduction we motivated this work by mentioning examples of behavioral architecture composition bugs that can be detected

using model-based analyses on component models. This section we describe how the information inferred using `ROSInfer` is used by these analyses to demonstrate the practical significance and usefulness of the inferred models motivated earlier. Combining behavioral component models with component-port-connector models, which can be inferred using an approach such as `ROSDiscover` [77] allows for analyses of intra-component-data-flow. Structural models alone do not contain information about how the inputs of a component are used and what is needed for the component to produce an output. Having input-output state machine models like the ones `ROSInfer` infers, allows to trace 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 [13] or logical reasoning [43].

Furthermore, synchronizing the resulting component state machines at their input/output messages allows for checking arbitrary Linear Temporal Logic (LTL) properties via approaches such as PRISM [50]. Thereby safety and security properties, such as the component changing to a desired state, no messages getting lost or ignored, or a component eventually publishing a certain message, can be checked [27, 25, 3].

Additionally, knowing the frequencies at which periodic messages get published allows the propagation of these frequencies to all transitive receivers of this data stream to check the desired frequency of message publishing further down the data stream and avoid unexpectedly high publishing frequencies.

3.5 Limitations

In this section we discuss the limitations of `ROSInfer` and differentiate which of those are intently caused by the approach versus limitations of the implementation that could be overcome with more engineering effort.

3.5.1 Limitations of the Approach. **Not using ROS API**: If developers of ROS systems implement reactive behavior without using the publish-subscribe API or periodic behavior without using the Rate API, the approach will not detect this type of behavior.

Other Triggers: If a behavior is triggered by other events than receiving a message in subscriber callback, such as reacting to messages from external devices received via serial ports[2], our approach cannot infer the trigger for this behavior.

Object logic: 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[3]. This dependency that is implicit due to complex logic within the imagine object cannot be inferred statically.

3.5.2 Limitations of the Implementation. **Input-dependencies**: Due to limited resources for implementing the approach and few cases of input-dependent behavior in the case study system, we ignored conditions on specific values of the inputs when implementing `ROSInfer`.

4 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.

4.1 Evaluation of the accuracy of `ROSInfer`

The main metric to evaluate the accuracy of `ROSInfer` is how well it can statically recover behavioral component models of real-world ROS systems. Our first research question is:

RQ1: How accurately does our implementation of the approach identify architecturally-relevant component behavior (reactive, periodic, state-based)?

Methodology for RQ1: Measuring accuracy requires some ground truth to which the result of the approach is to be compared to. Unfortunately, there is no reliable ground truth available for the architectural behavior of ROS components. Therefore, we needed to manually create a ground truth by reading the source code carefully and creating hand-written models of its behavior. Two of the authors of the paper split the work evenly. 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. Further, all 118 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.

After creating the handwritten models we execute `ROSInfer` on the source code and compare 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.

4.2 Evaluation of the core assumption of the approach

Our approach is based on the assumption that developers of ROS systems commonly use the ROS API to implement architecturally relevant component behavior that can fit in to the component model formalism we presented in this paper. So its accuracy is limited by the use of other means to implement reactive, periodic, and state-based behavior, even if a perfect static analysis was implemented.

[2]See example: https://github.com/autowarefoundation/autoware/blob/5c46036b02f08774a325c4929df121422ea73fab/ros/src/sensing/drivers/imu/packages/memsic/nodes/vg440/vg440_node.cpp#L406

[3]See example: https://github.com/autowarefoundation/autoware/blob/5c46036b02f08774a325c4929df121422ea73fab/ros/src/sensing/fusion/packages/points2image/nodes/vscan2image/vscan2image.cpp#L72

Since the C++ language and ROS API is too large to be covered entirely by a research prototype, an implementation of this approach will either be over-fitted to studied system or perform below what would be expected of an industrial product that implements the approach outlined in this paper. To evaluate the effectiveness of the general approach of using API-call-guided recovery of component-based behavioral in ROS systems, we posed the second research question:

RQ2: How do ROS developers implement architecturally relevant behavior?

Methodology for RQ2: Since we have access to the ground-truth information from the hand-written models and know how many of the behaviors can be found by ROSInfer from RQ1 we inspect the code for all behaviors that ROSInfer did not find and categorize them. There are cases in which the code uses a part of the ROS API or C++ language features that we have not covered in the implementation of ROSInfer yet but could reasonably be supported if more engineering effort is invested. There are other cases in which behaviors are implemented using very complex object logic that would require a further extension of the approach outlined in Section 3 or an extension of the model type. Therefore we measure how many behaviors could theoretically be found with a perfect implementation of the presented approach and formalism. The remaining cases violate the assumption that our approach is making and would motivate future work on extensions.

4.3 Case Study System: Autoware.AI

Autoware.AI [42] is the most popular open-source framework for self-driving vehicles. Since it is the largest and most complex ROS-based open source system, defining over 229 different components across more than 250 000 lines of code, it represents a large challenge for static analysis to infer the behavior adequately. With its first release in 2015 the system has been evolving over a long time and had to deal with many bugs that resulted from incorrect architectural composition [77]. Due to the large effort of manually creating ground truth models for the evaluation, we evaluated the approach on one system. However, we intentionally picked the system that contains the most complex logic and constitutes the largest challenge for static analysis to not bias the results in favor of the approach. It is not uncommon for recent top-tier software engineering research in robotics systems to feature only one case study system, due to the high effort of evaluation [22, 29].

4.4 Results

We evaluated ROSInfer on 106 components of Autoware.AI. The results are shown in Figure 5. For 12 components the static recovery process crashed due to run time errors in the implementation. These components are excluded from this evaluation.

4.4.1 RQ1. **Periodic Behavior:** Out of the 12 instances of periodic behavior in the ground truth ROSInfer successfully detected 6. For all of these cases the frequencies could be recovered correctly.

Hence, the precision for detecting periodic behavior is 50.00 %.

Reactive Behavior: Out of the 69 instances of periodic behavior in the ground truth ROSInfer successfully detected 46. **Therefore, the precision for detecting periodic behavior is 66.67 %.**

State-dependent Behavior: Out of the 20 state variables from the ground truth, ROSInfer recovered 8 (**40.00 %**). Of these, six cases could not recover the initial value, defaulting to ⊤. All of these ⊤ cases are easy to fix in implementation.

Out of the 27 state transitions from the ground truth, ROSInfer successfully recovered 13 (**48.15 %**). None of them include ⊤.

4.4.2 RQ2. **Periodic Behavior:** Out of the six cases of periodic behavior that ROSInfer does not find three cases are caused by the use of a different API for the sleep call (`std::sleep`, `std::this_thread::sleep_for`, or `usleep` instead of `ros::Rate::sleep`). Furthermore, two cases use a different API to publish a message (`tf::TransformBroadcaster::sendTransform`, which is a special case to publish transformations to the `tf` topic, instead of `ros::Publisher::publish`). Adding these API calls to the implementation would result in detecting these cases as well. Finally, one case could not be detected because publisher objects were passed as arguments to a separate message sending method. Since ROSInfer does not interprocedurally track the identify of publisher objects, it does not detect these cases of message sending. Support for this could be added with some engineering effort using object-oriented abstract interpretation. None of the false negatives of periodic behavior are caused by a limitation of the approach of API-based behavioral reconstruction. **Hence, we conclude that the precision of a perfect implementation of the approach for detecting periodic behavior on our case study system would be 100 %.**

Reactive Behavior: Out of the 23 cases of reactive behavior that ROSInfer does not find one case is caused by the use of a different API for registering a subscriber callback (`ros::Subscriber::registerCallback` instead of `ros::NodeHandle::subscribe`. Additionally, two cases use a different API to publish a message (`tf::TransformBroadcaster::sendTransform`. Further, seven cases could not be detected because publisher objects were passed as arguments to a separate message sending method. Similar to periodic behavior, these cases can be detected with additional engineering effort. Furthermore, three cases cannot be detected by ROSInfer because the component behavior is hidden in within a separate component class, which could be detected when investing more engineering effort into the static analysis.

Besides these limitations of the implementation, there are also limitations of the approach that would not benefit from simply improving the static analysis implementation. Of the remaining cases, seven cases use ROS message filters instead of single subscribers. With message filters, ROS developers can define a callback that is triggered when a set of messages arrive at different input ports at approximately the same time. Adding support for these cases requires not only to implement detectors for the message filter API but also changes in the formal model semantics, since it does not support a measure for approximately the same time of arrival. Finally, four cases include behavior that is triggered in response to an unsupported trigger (e.g., messages received via CAN bus, serial ports, or Mqtt). These eleven cases require a change of the model

	Periodic Behavior	Reactive Behavior	State Variables	State Transitions
Instances of that behavior	12	69	20	27
Recovery accuracy of ROSInfer (RQ1)	50.00 %	66.67 %	40.00 %	48.15 %
Optimal accuracy of the approach (RQ2)	100.00 %	84.06 %	55.00 %	66.67 %

Figure 5: The results of our case study evaluation on 106 components of Autoware.AI.

```
lane_planner::vmap::VectorMap all_vmap; // state variable

void cache_point(const vector_map::PointArray& msg)
{ //subscriber callback, similar callbacks exist for
     all_vmap.lanes and all_vmap.nodes
  all_vmap.points = msg.data; // state change
  update_values();
}

void update_values()
{
  if (all_vmap.points.empty() || all_vmap.lanes.empty()
     || all_vmap.nodes.empty()) // state condition
    return;
  [...]
  lane_planner::vmap::publish_add_marker([...]);
}
```

Figure 6: Simplified code snippet showing an example from 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 understand that the subscriber callback sets them to a state that would result in a change of the return value of the `empty` call.

semantics and therefore are limitations of the approach. **We conclude that the precision of a perfect implementation of the approach for detecting reactive behavior on our case study system would be 84.06 %.**

State-Variables: Out of the 12 state variables that cannot be recovered, 6 cases contain complicated object logic, such as whether a list or map is empty. Figure 6 shows an example of this[4] 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. Further, three cases could not be recovered because the base behavior was not recovered due to a limitation of the approach for this base behavior. Finally, three cases were not recovered because of a limitation in the implementation to recover the base-behavior and therefore are the only cases of state variable recovery that can be attributed to imperfect static analysis. **We conclude that the precision of a perfect implementation of the approach for detecting state variables on our case study system would be 55.00 %.**

State Transitions: All of the 14 cases of unrecovered state transitions are caused by unrecovered state variables. When tracing the root-cause for not recovering the dependent state variables, five cases are caused by a limitation of the approach while nine cases are caused by a limitation of the approach. **We conclude that the precision of a perfect implementation of the approach for detecting state transitions on our case study system would be 66.67 %.**

Lessons Learned about ROS Components: When building the behavioral models for ROS components we noticed a few patterns:

(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 2.
(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. This makes the code more extensible and easier to read by human developers, but harder to analyze using static analysis.

4.5 Threats to Validity

4.5.1 Internal Validity. **Human error during model creation:** The ground-truth models were created 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.

4.5.2 External Validity. **Selection of Case Study System:** 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 Autoware.

5 DISCUSSION

In this section we discuss considerations from the practical software engineering context that affect the effectiveness of the approach, how the unavoidable incompleteness of recovered models will impact the usefulness of the models, and how the execution time of the analyses scales to complex real-world systems.

[4]Simplified and annotated from `waypoint_clicker` node source: https://github.com/autowarefoundation/autoware/blob/5c46036b02f08774a325c4929df121422ea73fab/ros/src/computing/planning/motion/packages/waypoint_maker/nodes/waypoint_clicker/waypoint_clicker.cpp#L71

5.1 Effectiveness in Practice

The real-world effectiveness of the approach mainly depends on two factors: (a) the usage of framework APIs to implement architecturally relevant behavior and (b) the complexity of the programming language constructs that connect the API calls and their parameters.

The presented approach cannot recover the behavior of components that implement their behavior without using the framework API. In practice this can happen if developers are unfamiliar with the provided API, if the API is too complex so that developers chose to use simpler constructs, or if the API is too simple so that developers chose to implement a more complex logic that fits their needs [61, 32, 83, 69].

Even though looking for API calls and their connections makes the problem of behavioral reconstruction significantly easier, the remaining problem is still undecidable in the general case. Examples of dynamic programming language features used to implement architecturally relevant behavior we observed in ROS systems are storing publishers and/or subscribers in arrays or lists rather that single variables, storing the values of state variables in dictionaries rather than state variables, passing publisher objects or rate objects as parameters to functions, or using complex object logic in conditions, similar to the example shown in Figure 6. These dynamic language features are out of scope of a static analysis for behavioral recovery. Hence, component implementations that use these language features will not be recovered correctly.

5.2 Known Unknowns / Incomplete Models

The resulting models can be incomplete in practice. 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 $\top$ that represents 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, initial values or other assignments of state variables, or values that state variables are compared to in conditions. They occur when other variables are references that are not tracked by the static analysis, 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 easier 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 the 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.

Unknown unknowns are more limiting in practice, since the resulting model might seem reasonable while missing some information. They occur when developers do not use a common API to implement component behavior, or when they do not follow the patter that `ROSInfer` is looking for. They 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. Since incomplete models would only result in false positives for the analyses that we mentioned in Section 3.4, incomplete models would not give architects a false sense of security. In this case, having incomplete models would still be preferable to having no models, because even incomplete models might be able to find some behavioral architecture composition bugs that would not have been found otherwise.

5.3 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 architectural recovery of component models took on average 31.73 s. The fully automated analysis of the entire Autoware.AI system took 121.10 min. This should demonstrate that the static analysis scales to real-world systems and could integrate well into iterative software development practices. The effort it took to create handwritten models can be approximated with about 120 work hours of manual labor[5]. 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 and economical.

6 RELATED WORK

To demonstrate the novelty of this work, we discuss other analyses that have been performed on robotics systems, approached to recover static architectures, and dynamic analyses of behavioral models and argue how our work different 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 [53, 5, 35, 66]. For example, the systems Phriky [65], Phys [41], and Physframe [40] use type checking to find inconsistencies in assignments based on physical units or 3D transformations in ROS code.

Furthermore, Swarmbug [37] 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.

[5]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.

6.2 Recovery of Structural Architectures

Most approaches for static recovery of software architectures reconstruct structural views of software modules from the perspective of a developer [74, 67, 9, 33, 23, 55, 58, 57, 4, 19, 24, 17]. 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 in [30, 75, 26]. Since they show the code before compiling it, the module view does not show the relationships of components during run time [18].

ROSDiscover [77], HAROS [72, 70], and the tool by Witte et al. [82] can reconstruct component-port-connector (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.

6.3 Dynamic Recovery of Behaviors

Behavioral models of components can be inferred using dynamic analysis by observing the component behavior of representative execution traces. For example, DiscoTect [73] and Perfume [64] construct state machines from event traces. Similar approaches also use method invariants [48], LTL property templates [52] to increase the effectiveness. Domain-specific approaches have been proposed for for CORBA systems [60] or telecommunication systems [59] 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 control-dependent. Furthermore, dynamic approaches require either an accurately simulated execution environment or access to the real-world special-purpose 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.

7 CONCLUSIONS

In this paper we have shown that looking for specific API calls of the ROS framework makes the challenging and undecidable problem of statically reconstructing behavioral models from C++ source code more tractable by exploiting observations of common practices of the ROS ecosystem. 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. While this paper focused only on ROS-based systems, we believe that API-call-based recovery 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.

We envision the following future work to be enabled by our contributions:

Overcoming Limitations of Implementation and Approach: The most immediate future work is to overcome the engineering limitations mentioned in Section 4.4.2. As shown there, adding more API calls and increasing the support of static analysis for inter-procedural object-oriented analysis can increase the precision of `ROSInfer`. Furthermore, research on enhancing the formalism of the approach to support more types of triggers and to capture more complex state machines of the behavior of object-oriented systems is needed to further improve the accuracy of static recovery.

Combination of Static and Dynamic Analysis: As the results from RQ2 have shown, even perfect static analysis still leaves incomplete models in some cases. Furthermore, static analysis cannot infer execution times of tasks, producing models that cannot be used for most kinds of performance analysis, bottle neck analysis, or analysis of race condition. 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 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.

Support for Evolution of hand-completed Models: One main advantage of automated architectural recovery is that it is cheaper to keep it up-to-date for evolving software, because it can be executed periodically or after major changes to the architecture. Since the models created by static recovery can never be guaranteed to be complete, developers need to manually complete them for practical use. To not override the hand-completed models for every re-generation, future work is needed on separating the handwritten parts from the generated parts of the model.

Human-subject Evaluation: In this work we have not quantified the effort of an average robotics software engineer that would be saved by using `ROSInfer` instead of manually inferring these models. Future research that measures the time it takes practitioners to infer the models for scratch and complete the resulting models from `ROSInfer` to calculate the difference is needed to make stronger claims about the impact on practice.

Generalization to other Frameworks: The approach of API-call-based recovery of component behavior is not inherently specific to the ROS framework. Other component frameworks, such as NASA FPrime [10], that provide APIs for component interaction mechanisms could implement this approach as well.

8 DATA AVAILABILITY

The entire implementation of `ROSInfer` as extension to ROSDiscover [77], all handwritten models used for evaluation, the automatically inferred models with `ROSInfer`, and analysis scripts are included in the replication package and will be made publicly available.

REFERENCES

[1] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. 2020. A Study on Challenges of Testing Robotic Systems. In *International Conference on Software Testing, Validation and Verification (ICST)*, 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, 16–39. DOI: https://doi.org/10.1016/j.jss.2016.08.039.

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

[4] 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.

[5] Hugo Araujo, Mohammad Reza Mousavi, and Mahsa Varshosaz. 2022. Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review. *ACM Trans. Softw. Eng. Methodol.*, (June 2022). Just Accepted. DOI: 10.1145/3542945.

[6] 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. DOI: 10.3390/jsan10030048.

[7] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage. 2006. Performance prediction of component-based systems. In *Architecting Systems with Trustworthy Components*. Springer Berlin Heidelberg, 169–192. DOI: 10.1007/11786160_10.

[8] 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.

[9] 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.

[10] 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/.

[11] 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.

[12] 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.

[13] 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.

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

[15] 2007. *Trends in Component-Based Robotics. Software Engineering for Experimental Robotics*. Springer Berlin Heidelberg, 135–142. DOI: 10.1007/978-3-540-68951-5_8.

[16] 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)*, 33–38. DOI: 10.1145/3526071.3527521.

[17] 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.

[18] 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.

[19] 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.

[20] 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. DOI: https://doi.org/10.1016/j.promfg.2017.07.107.

[21] 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: https://doi.org/10.1016/j.cola.2020.101021.

[22] Yao Deng, Xi Zheng, Mengshi Zhang, Guannan Lou, and Tianyi Zhang. 2022. Scenario-based test reduction and prioritization for multi-module autonomous driving systems. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM, 82–93. DOI: 10.1145/3540250.3549152.

[23] 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.

[24] 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.

[25] Xiaocheng Ge, Richard F. Paige, and John A. McDermid. 2010. Analysing system failure behaviours with prism. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, 130–136. DOI: 10.1109/SSIRI-C.2010.32.

[26] 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.

[27] 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 Berlin Heidelberg, 625–639.

[28] 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)*, 44–50. DOI: 10.1109/FormaliSE.2017.9.

[29] Ruidong Han, Chao Yang, Siqi Ma, JiangFeng Ma, Cong Sun, Juanru Li, and Elisa Bertino. 2022. Control parameters considered harmful: detecting range specification bugs in drone configuration modules via learning-guided search. In *International Conference on Software Engineering (ICSE '22)*. ACM, 462–473. DOI: 10.1145/3510003.3510084.

[30] 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.

[31] 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.

[32] Daqing Hou and Lin Li. 2011. Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions. In *2011 IEEE 19th International Conference on Program Comprehension*, 91–100. DOI: 10.1109/ICPC.2011.21.

[33] 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.

[34] 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.

[35] 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.

[36] 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: https://doi.org/10.1016/0376-6349(87)90023-X.

[37] 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.

[38] 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.

[39] 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*, 6107–6112. DOI: 10.1109/IROS.2010.5652394.

[40] 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.

[41] 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.

[42] 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.

[43] Mourad Kmimech, Mohamed Tahar Bhiri, and Phillipe Aniorte. 2009. Checking component assembly in acme: an approach applied on uml 2.0 components model. In *2009 Fourth International Conference on Software Engineering Advances*, 494–499. DOI: 10.1109/ICSEA.2009.78.

[44] Sophia Kolak, Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

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

[52] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL Specification Mining (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 81–92. DOI: 10.1109/ASE.2015.71.

[53] 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.

[54] 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.

[55] S. Mancoridis, B.S. Mitchell, Y. Chen, and E.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.

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

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through Hoops: Why Do Java Developers Struggle with Cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering* (ICSE '16). ACM, 935–946. DOI: 10.1145/2884781.2884790.

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Martin P Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering*, 16, 6, 703–732. DOI: https://doi.org/10.1007/s10664-010-9150-8.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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 *Proceedings of the 19th IEEE International Conference on Software Architecture*. (ICSA '22). IEEE, 112–123. DOI: 10.1109/ICSA53651.2022.00019.

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

[79] Milos Vasic and Aude Billard. 2013. Safety issues in human-robot interactions. In *International Conference on Robotics and Automation*, 197–204. DOI: 10.1109/ICRA.2013.6630576.

[80] 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.

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

[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] Minhaz F. Zibran, Farjana Z. Eishita, and Chanchal K. Roy. 2011. Useful, But Usable? Factors Affecting the Usability of APIs. In *2011 18th Working Conference on Reverse Engineering*, 151–155. DOI: 10.1109/WCRE.2011.26.