ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems

Christopher S. Timperley, Tobias Dürschmid, Bradley Schmerl, David Garlan, and Claire Le Goues
Carnegie Mellon University, Pittsburgh, PA, USA
{ctimperl, tdurschm, schmerl, garlan, clegoues}@cs.cmu.edu

***Abstract*—Robot systems are growing in importance and complexity. Ecosystems for robot software, such as the Robot Operating System (ROS), provide libraries of reusable software components that can be configured and composed into larger systems. To support compositionality, ROS uses late binding and architecture configuration via "launch files" that describe how to initialize the components in a system. However, late binding often leads to systems failing silently due to misconfiguration, for example by misrouting or dropping messages entirely.**

In this paper we present `ROSDiscover`, which statically recovers the run-time architecture of ROS systems to find such architecture misconfiguration bugs. First, `ROSDiscover` constructs component level architectural models (ports, parameters) from source code. Second, architecture configuration files are analyzed to compose the system from these component models and derive the connections in the system. Finally, the reconstructed architecture is checked against architectural rules described in first-order logic to identify potential misconfigurations.

We present an evaluation of `ROSDiscover` on real world, off-the-shelf robotic systems, measuring the accuracy, effectiveness, and practicality of our approach. To that end, we collected the first data set of architecture configuration bugs in ROS from popular open-source systems and measure how effective our approach is for detecting configuration bugs in that set.

***Index Terms*—ROS, Software Architecture, Static Analysis**

I. Introduction

Ensuring that robotics systems are free from defects is a pressing research challenge. Robots are ubiquitous, with applications ranging from warehouses and autonomous delivery to manufacturing vehicles and driving them. Since robots often interact with humans during their tasks, defects in robotics systems can cause significant safety hazards.

Robot systems are often component-based systems, and can be comprised of hundreds of different components [1], [2]. To construct a robot system, developers define and configure component parameters and connect components so that they can communicate in a variety of ways, such as through topics (publish-subscribe), service calls (synchronous call-return), and actions (asynchronous call-return).

The most popular open-source framework for component-based robotics systems, the Robot Operating System (ROS), uses late binding mechanisms for dynamic composition of components. To increase the reusability of the more than 6 000[1] software packages in the ROS ecosystem, ROS uses configuration mechanisms and connectors that are not bound during compile time [3]. These mechanisms include string-based identifiers for topics, services, actions, and parameters, as well as remappings between these identifiers [4].

While useful, the use of late binding for component composition is prone to misconfiguration of component connections, such as mismatching names or types. This can result in messages being misrouted or dropped, configuration parameters being silently ignored, or unexpected exceptions [5]. Such bugs can be hard to find since they only manifest at run time.

Our key insight into identifying these bugs is that many of them manifest as errors in the run-time architecture. That is, they result from inconsistent composition or configuration of components and connectors. We call this class of bugs *architecture misconfiguration bugs*. Run-time architectural models contain components (incl. typed parameters and ports), and connectors. Given a set of architectural well-formedness rules, architectural checking could in principal identify many of these bugs [6]. However, ROS run-time architectures are not specified in one place. Instead, they are typically implicitly defined by widely scattered pieces of source code and configuration files, rather than via formal architectural models [7]. This motivates the use of automatic techniques to reconstruct run-time architectures to support automatic analysis and identification of architecture misconfiguration bugs.

General techniques to reconstruct arbitrary run-time architectures from source code are not accurate enough to reliably capture component connections and configurations, especially for publish-subscribe [8], [9]. Run-time architectures can differ significantly from the source code structure, and the variety of implementations makes it hard to infer connectors absent domain-specific assumptions. This limits the accuracy (i.e., the precision and recall of recovered elements) of generic static reconstruction approaches. Existing ROS-specific techniques [10]–[12] have achieved higher accuracy by exploiting connections between ROS Application Programming Interface (API) calls to connector usage. These prior approaches are still inadequately accurate for reliable automatic identification of misconfigurations, because they do not leverage sufficient source level information. However, their initial success supports our intuition that such domain-specific knowledge is critical to effective architectural recovery.

In this paper, we present `ROSDiscover`, an approach for statically recovering run-time ROS architectures from source code and configuration files to identify architectural miscon-

This work has been supported in part by NASA (Award 80NSSC20K1720), AFRL (Award 19-PAF00747), and the NSF (Award CCF-1750116)

[1] https://index.ros.org/stats/ [Date Accessed: 4th November 2021]

figuration bugs. First, ROSDiscover constructs component models (typed ports and parameters) via static analysis that focuses on a small set of calls to the ROS API. Second, architecture configuration files are analyzed to compose the system from these component models and derive the connections in the system. Finally, the reconstructed architecture is checked against architectural rules described in predicate logic to identify potential misconfigurations that could cause defects.

Our approach is effective, accurate, and scalable, exploiting several key observations about component-based robotics sytems in general and ROS systems in particular:

1) Component architectures are defined via API calls that have well-understood architectural semantics [4].
2) The composition and configuration of components to build larger systems is done in separate architecture configuration files (i.e., launch files). Most of these result in "quasi-static" systems. That is, architectures rarely change following run-time initialization [4].
3) ROS systems rely on a small de facto core library of components [13]. We provide a small number of handwritten models for components from this library that are too complex or dynamic to be statically recovered.

Overall, we make the following contributions:

- An approach to statically recover run-time architectures in ROS systems based on symbolic execution of API calls and static analysis of configuration files;
- An approach to use first-order logic to find architecture misconfiguration bugs in recovered architectures;
- ROSDiscover, an open-source implementation of our approach: https://github.com/rosqual/rosdiscover;
- The first available data set of 29 architecture misconfiguration bugs across 5 open-source ROS system;
- An evaluation on popular real-world open source ROS systems and the bugs from our newly collected data set.

Our theoretical approach is generalizable to other ecosystems for which above mentioned observations hold true, such as flight systems [14] or microservice architectures [15]. The corresponding artifact containing ROSDiscover, our data set, Docker images, analysis scripts, and results can be found here: https://doi.org/10.5281/zenodo.5834633.

II. Architecture Misconfiguration Bugs in ROS

We begin by defining *architectural misconfiguration bugs* and describe how they relate to the ROS framework.

A. Background on ROS

From a software perspective, a ROS-based system is composed of components called *nodes*, which are processes that perform computation.[2] ROS 1[3] provides a *ROS Master*, a centralized hub that provides naming and registration for all nodes in a ROS system and parameter server (a shared dictionary of names and values) that stores configuration parameters. Nodes use the parameter server to configure themselves dynamically. In our context, a component parameter is a value passed in on component launch that can customize its behavior. For our purposes, the most interesting uses of parameters focus on configuring topic or service names, since those are relevant to architectural interactions between nodes.

ROS nodes interact with each other through a set of ROS-standard communication styles: topics, services, and actions.

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

Services implement a synchronous call-return style of communication between nodes. Nodes attempting to call a service look up the service provider in a registry based on the string-based name of the service. Because in ROS 1 they are synchronous, services are intended for short queries, such as the state of a node or short mathematical computation.

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

To initialize and configure a ROS system, ROS 1 allows developers to specify *launch files* – Extensible Markup Language (XML) files that specify which nodes to start, node parameters, and topic name remaps when composing nodes. Launch files often include other launch files and can contain conditional logic. Note that launch files do not directly connect nodes with each other, since this is done using ports defined within components.

The configuration of nodes and their connections can be observed during run time by traversing the ROS graph, a tool to query the current run-time architecture of a running system.

B. Architecture Misconfiguration Bugs

We call bugs that result from an inconsistent composition of software components *architecture misconfiguration bugs*. In this context, *composition* refers to the parameterization or configuration of components or connectors. Such a composition can be *inconsistent* in the following ways:

- A connector fails to connect ports it is supposed to connect (e.g., a topic name mismatch).
- A connector connects ports of inconsistent types (e.g., message type mismatch between what the publisher sends versus what the subscriber receives).
- A mandatory component parameter has not been set.
- A component parameter has been set to an instance of an incompatible type.

[2] See https://wiki.ros.org/Nodes. *Nodelets* are specialized nodes that can be loaded to share memory; we include them under the umbrella term "Node".

[3] We focus on ROS 1 because doing so gives us access to significantly more systems and historical defects (ROS 2 was released in 2020)

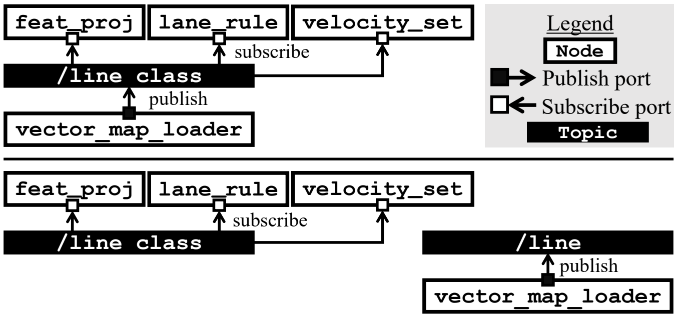


Fig. 1: Component-connector views of a publish-subscribe connector in Autoware.AI before (top) and after (bottom) a bug was introduced. Renaming the topic line_class to line only in the publisher disconnected publishers and subscribers accidentally. The publishers and subscribers are dangling.

Architecture misconfiguration bugs often arise due to misspellings or when large systems evolve inconsistently, as Facebook's multi-million-dollar incident in October 2021 shows [16]. For example, while developers change names or types of parameters and topics in one part of the system, they might forget to change them in other parts of the system. To illustrate, consider Figure 1, illustrating a bug in Autoware.AI, the most popular open-source framework for self-driving vehicles. The system contains the vector_map_loader node, which publishes the line_class, point_class, area_class, pole_class, and box_class topics. The nodes feat_proj, lane_rule, and velocity_set subscribe to some of these topics, as shown at the top of the figure. In one commit, the developers updated the topic names to drop the "_class" suffixes.[4] However, they forgot to change these topic names in the feat_proj, lane_rule, and velocity_set nodes, as illustrated in the bottom of the figure; they therefore did not receive any data. It took the developers two months to find and fix the bug.[5]

In general, to check whether a given version of the architecture has misconfiguration bugs, one needs to know which components are in the systems and how they are connected. A simple string search for API calls and their arguments is usually not sufficient for this, because many arguments do not get passed to the APIs inline or as literals and the context of the API call is relevant to determine the name space [4].

Hence, we propose a static analysis approach to find bugs like these, overcoming the limitations and expense of testing complex robotics systems [17], [18]. Integrated into a continuous integration pipeline, our tool could tell the developers that their architecture is inconsistent when pushing a new commit.

C. A Data Set of Architecture Misconfiguration Bugs in ROS

To illustrate the variety of architecture misconfiguration bugs and to foster more research on them, we construct and provide a data set of bugs from real-world open-source ROS systems. The data set can be found in the artifact.

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

TABLE I: Statistics on the systems contained in our bug data set.

Selection Criteria: We collected documented bugs on GitHub from repositories discussed in Malavolta et al. [19] (as these repositories are well-studied and mature). For each repository, we searched for the key words "topic bug", "topic fix", "subscribe bug", "subscribe fix", "publish bug", "publish fix", "topic rename", "launch file fix", and "launch file bug" in commits, issues, and pull requests. Not all of the results refer to run-time architecture misconfigurations, and so we manually verified/filtered the bugs by inspecting the code, change history, and documentation. We excluded bugs for which we were unable to compile the software versions. As shown in Table I, the data set contains 29 bugs across 5 systems. Note that this is not intended to be a complete list of architecture misconfiguration bugs in those systems.

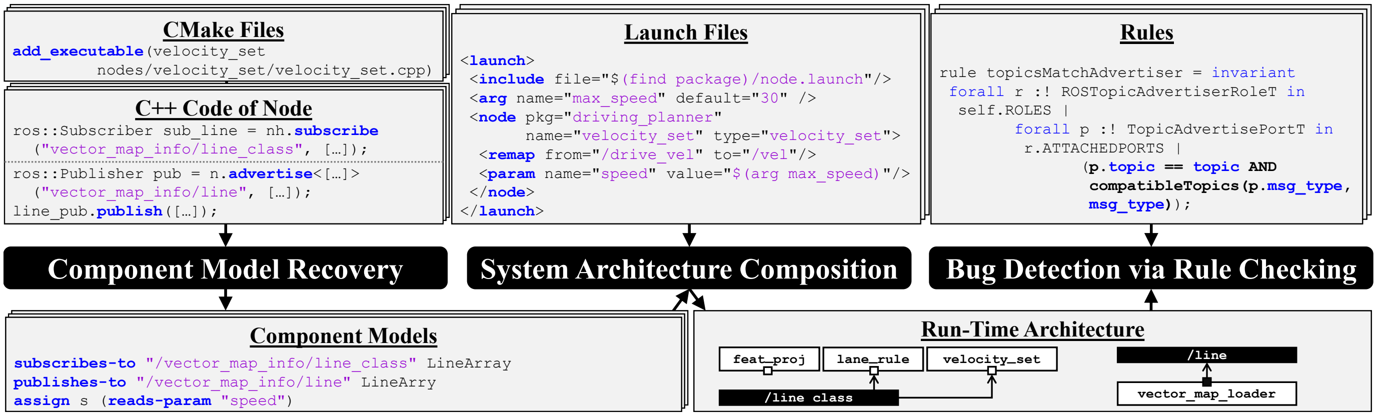
Collected Data: For bugs caused by a broken publish-subscribe connector (i.e., inconsistent topic names or message types), we identified the publisher, topic, subscriber, and a set of launch files that launch the corresponding nodes. For bugs caused by a wrong configuration (i.e., inconsistent parameter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

Building Historic Project Versions: Some of the commits related to the bugs are many years old and were built with much older versions of their dependencies and much older versions of ROS (the oldest bug dates back to March 2014 and was reproduced with ROS Indigo Igloo). To support replicability, we created Docker images for each commit, each containing the versions of their build- and run-dependencies (including ROS packages, CUDA API version, external libraries, compilers, and the ROS distribution). If the project documented which versions of a dependencies were used, we installed these in the Docker image. Otherwise, we installed the most recent version at the time of the corresponding commit date.[6] For versions for which we were unable to construct the Docker images according to this methodology, we forward-ported the bugs (i.e., applied the bug-introducing change to a version of the software that we can build).

[4] https://github.com/Autoware-AI/autoware.ai/commit/fc8f69

[5] https://github.com/Autoware-AI/autoware.ai/commit/c2a090

[6] Using https://github.com/rosin-project/rosinstall_generator_time_machine



```
add_executable(velocity_set
         nodes/velocity_set/velocity_set.cpp)
```

```
ros::Subscriber sub_line = nh.subscribe
  ("vector_map_info/line_class", […]);
ros::Publisher pub = n.advertise<[…]>
  ("vector_map_info/line", […]);
line_pub.publish([…]);
```

```
<launch>
 <include file="$(find package)/node.launch"/>
 <arg name="max_speed" default="30" />
 <node pkg="driving_planner"
       name="velocity_set" type="velocity_set">
  <remap from="/drive_vel" to="/vel"/>
  <param name="speed" value="$(arg max_speed)"/>
 </node>
</launch>
```

```
rule topicsMatchAdvertiser = invariant
 forall r :! ROSTopicAdvertiserRoleT in
  self.ROLES |
       forall p :! TopicAdvertisePortT in
        r.ATTACHEDPORTS |
              (p.topic == topic AND
               compatibleTopics(p.msg_type,
               msg_type));
```

```
subscribes-to "/vector_map_info/line_class" LineArray
publishes-to "/vector_map_info/line" LineArry
assign s (reads-param "speed")
```

Fig. 2: Overview of the analysis stages (dark rectangles), data (rectangles), and data flow (arrows) of ROSDiscover. First, the component interface models are inferred from source code. Then, these models are composed according to architectural configuration files. Finally the architecture is checked against well-formedness rules specified in first-order logic.

III. ROSDiscover

A common approach to detect bugs that manifest in the run-time architecture, is to check well-formdness rules on run-time architectural models. This section describes the theoretical foundation of static recovery of run-time architectures of ROS systems to find architecture misconfiguration bugs.

Our architectural recovery must reconstruct the structural connections between components, as well as component parameters. In architectural terms, *structural connections* correspond to the ports and the connectors between them. Meanwhile, component parameters control (among other things) the naming of component interfaces, such as topics or services. Beyond being undecidable in general, effective static recovery of run-time architectures is difficult. First, a system's run-time view can differ from the more accessible module view. Moreover, architectural connectors and ports can be implemented in a variety of ways, precluding a domain-independent approach to identifying and modeling them. We overcome these challenges for ROS by relying on three key observations:

1) **Well-defined Component Framework API:** ROS run-time architectures are defined via a small set of API calls that have simple and well-understood architectural semantics [4]. Static analysis for component port recovery can focus on a small space of ROS API calls.
2) **Quasi-static Architectures:** ROS uses separate architecture configuration files (launch files) to compose components into larger systems. The resulting systems are typically “quasi-static” [4], rarely changing after initialization. Architectural recovery can thus focus on parsing launch files rather than considering more complicated behavioral change. For (rare) dynamic architectures, the output contains elements than are not in the ground truth.
3) **Small Core Library:** ROS systems heavily rely on a small de facto core library of packages, most from the core library [13]. Core packages can be more difficult to analyze statically because they often include complex control flows and extension mechanisms to support generality. We support complex systems by providing a way to specify reusable handwritten models for components that are difficult to analyze statically.

Figure 2 provides an overview of the analysis process and its inputs. The analysis consists of three parts:

1) The *Component Model Recovery* stage recovers a component's interface and internal behavior based on the component's source code (Section III-A).
2) The *System Architecture Composition* stage creates a component-port-connector model of an entire system configuration using the models from the previous stage and the architecture configuration files (Section III-B).
3) The *Bug Detection* stage finds architecture misconfiguration bugs by checking well-formedness rules specified in first-order logic on the system model (Section III-C).

In this paper we focus on the recovery of topic, service, and action ports for components and bugs that results from a misconfiguration of their corresponding connectors. To reduce implementation effort, we limit the analysis to C++ rather than Python because C++ is the most used language in ROS [4]. Static analysis of this kind in Python is possible and part of our future engineering effort. The inputs to ROSDiscover consist of ROS launch files, CMake files, package descriptions, and C++ source code files corresponding to a given system.

A. Component Model Recovery

The *Component Model Recovery* stage creates a symbolic description of the interface of each individual component from node source code. Each description, or *component model*, specifies the ports (i.e., publish, subscribe, service calls, and provided services) and parameters of its associated component. This information is represented as a set of symbolic function summaries. Each summary describes the architectural effects of an associated function in terms of its arguments, represented symbolically, as described below.

To find all calls to architecturally-relevant APIs in a component, we use standard static analysis. ROSDiscover identifies the source files in a component by analyzing associated

```
PoseToTF(ros::NodeHandle nh, std::string rtopic)
{
  std::string pose_topic_name;
  nh.getParam("pose_topic", pose_topic_name);
  nh.subscribe(pose_topic_name, 10,
    &PoseToTF::callbackGetPose, this);
  nh.advertise<std_msgs::Empty>(rtopic, 1);
}
```

(a) simplified C++ source code

```
func PoseToTF(nh: NodeHandle, rtopic: String) {
  pose_topic_name := readParam(joinNamespace(nh,
    "pose_topic"), "string");
  subscriber(joinNamespace(nh, pose_topic_name),
    "geometry_msgs/Pose");
  call("PoseToTF::callbackGetPose");
  publisher(joinNamespace(nh, rtopic)), "std_msgs/Empty");
}
```

(b) corresponding symbolic function summary

Fig. 3: `ROSDiscover` recovers component interfaces by producing and composing symbolic summaries for individual source code functions, describing the architecturally relevant API and function calls within those functions.

CMake files and build output, and using off-the-shelf Clang functionality to create/merge the Abstract Syntax Trees (ASTs) for all component translation units. This AST can be traversed to find calls to API functions defining the run-time component interface. For publish-subscribe connectors (i.e., topics), this includes the `advertise`, `publish`, and `subscribe` API methods. For call-return connectors (i.e., services), this includes `call` and `advertiseService`. For parameters, this includes the `param`, `getParam`, and `setParam`.

However, this AST analysis is inadequate to fully describe the component interface. Figure 3 illustrates: the code on the left-hand side calls `subscribe`, an architecturally-relevant API function. However, `pose_topic_name`, the topic name passed to `subscribe` is defined by a parameter. The topic name provided to the `advertise` API call on the following line is given by a function argument, `reset_topic`. Moreover, the call is made through a node handle object, `nh`, which specifies and controls the call's namespace.

Thus, the full component model is composed of summaries of all functions that can reach an architecturally-relevant API call, thereby providing necessary context around the parameters. Standard reachability analysis (supported by Clang) identifies relevant functions (those that can reach a relevant API call). `ROSDiscover` lifts each relevant API or function call into a statement in the symbolic summary, capturing architecturally relevant arguments to that call. Topic names and message formats are recorded for publishers. For function calls, only the arguments that are used in the callee's symbolic summary are captured. Our implementation captures strings and node handles, but ignores other data types (e.g., integers, floats, vectors) as they are seldom used in API calls. Call arguments are replaced with symbolic values computed in relatively standard ways. String literals are converted to literals in the summary; the effects of common ROS and C++ operations and library functions (e.g., concatentation) are modeled; variable references are traced to their reaching definitions, where arguments to the function call and the results of ROS parameter reads are represented symbolically.

Figure 3b provides an example of a recovered function summary. The first statement models the effect of reading a parameter, `readParam(..)`, and storing its value to `pose_topic_name`. The qualified parameter name is obtained by resolving the unqualified name `"pose_topic"` in its associated namespace (`nh`) via `joinNamespace`. For example, if the namespace is `/robot`, then the qualified name of the parameter accessed from the parameter server is `/robot/pose_topic`.] The next statement uses the value of the parameter to define a new subscriber via `subscribe`. Finally, a call is made to an architecturally relevant function via `call`, and a publisher is defined using a topic name provided by a function argument, `reset_topic`.

Handwritten Models: Our symbolic execution implementation balances expressiveness with scalability and cannot statically infer or model all behavior. For example, some ROS systems specify custom configuration files with an application specific format; this, among other sources of imprecision (like dynamically linked libraries), is out of `ROSDiscover`'s present scope. `ROSDiscover` by default represents such information using $\top$ ("top"), an overapproximation. We support the ability to manually refine analysis results by providing handwritten models. If a component is reusable because it is part of the ROS core library, associated models can be reused as well to minimize specification overhead; we specified 15 such models as part of our evaluation. They are written in an embedded DSL for Python filling in a data structure. On average, each model has 28.78 lines of code, ranging from 5 to 170. The models can be found in the artifact.

B. System Architecture Composition

The *System Architecture Composition* stage uses the system's architecture configuration files to compose the inferred component models (Section III-A) into a component-port-connector model of the whole system. Overall, this involves *instantiating* the symbolic values in the component models by resolving them to their concrete values. To illustrate why symbols in component models cannot be fully resolved prior to composing them, consider topic remappings, a common practice [4]. A developer may provide code to implement a camera processing node that subscribes to a topic called `camera`. A robot may have that publish images to topics `left_camera` and `right_camera` respectively. The resulting run-time architectural model should contain two instances of the camera processing node, for each set of images. The launch file will remap the `camera` topic of the first `camera` instance to subscribe to the `left_camera`, and the second instance to `right_camera`. The one component can therefore be configured multiple ways at run time.

The composition stage thus recursively parses the system launch files to determine (a) which nodes are run/loaded, (b) how they are parameterized, and (c) how topics are remapped at run time. We use the information from the node tag in the launch files to identify the names and types of launched/loaded nodes. Launch files also describe parameters used to configure components and topics. Publish-subscribe connectors are reconstructed based on the topic names from component models. In ROS, topic names are unique. Hence, for each publisher and subscriber that refer to a topic with the same name, we create a publish-subscribe connector. Call-return connectors are reconstructed based on the service calls from the component models, each of which creates a call-return connector. The caller role is set to the component instance initiating the call. The service provider role is set to the component instance that provides this service (a service can only have one provider in ROS).

C. Bug Detection via Rule Checking

The *Bug Detection* stage takes the component-connector model of the system architecture as an input and checks a set of rules described in first-order logic. Each component-connector system model is a typed directed graph that connects component instances via connectors. A typical approach in architectural checking describes predicates on those graphs and defines assertions on those predicates describing the architectural design intent [6], [20], [21].

We use Acme [22] to check the rules and report all rule violations. We developed a ROS Acme style, which can be found in the ROSFam.acme in the artifact, to describe a well-formed ROS architecture. These have on average 2.6 lines of code and are general to all of ROS, and so need to be defined only once. It is possible for domain experts to define additional rules for classes of robot system or at the system level, but this has not been explored in this paper.

The main challenge of bug detection is to define rules that find useful errors while minimizing false positives [23], [24]. To strike a balance, we picked these ROS style rules for detecting misconfigured topics in ROSDiscover:

1) There should not be connectors that connect ports of different message types (rule shown in Figure 2).
2) There should not be a dangling subscriber/publisher (i.e., a subscriber without publisher or vice versa) if there is a publisher/subscriber that has the similar type and a similar topic name.
3) There should not be a dangling publisher if there is a dangling subscriber that has a similar type.
4) There should not be a dangling subscriber if there is a publisher of a similar type.

To identify similar names or types we measure the edit distance between identifiers. We use a variation of Damerau–Levenshtein distance [25], [26] that considers that developers often use underscores in identifiers, and topic names use slashes to denote name spaces. We adjust the cost of insertion and substitution to be the distance metric between the segments split by "/" and "_".

D. Implementation

ROSDiscover works on containerized versions of robotics systems, simplifying extensibility and replicability. A layer called ROSWire manipulates ROS-specific files and processes on ROS Docker containers. We provide basic scaffolding for the static analysis (e.g., reachability). The system is written in Python, comprising about 15 000 lines of code. In terms of performance, node recovery is the most expensive operation because it must do static analysis, taking 47 sec per node on average. However, this only needs to be done once per component. Component models are modeled using Acme [22]. Once recovered, constructing and checking that model takes about two minutes.

IV. Evaluation

We designed ROSDiscover to trade off between false positives, false negatives, and efficiency, with a goal of being practical for typical ROS systems. To evaluate ROSDiscover, we address the following research questions:

- **RQ1:** *How accurately does ROSDiscover statically recover the interface of ROS nodes?* This question addresses the accuracy of *Component Model Recovery*.
- **RQ2:** *How accurately does ROSDiscover statically recover run-time architectures of real ROS systems?* This question addresses the accuracy of the *System Architecture Composition*.
- **RQ3:** *How effectively does ROSDiscover find configuration bugs in real ROS systems?* This question evaluates the practical use of our architectural recovery approach as applied to finding architecture misconfiguration bugs by measuring false positives and false negatives.

Data sets: RQ1 and RQ2 require a set of stable ROS systems, while RQ3 requires a set of known buggy versions of ROS systems. We evaluate RQ3 using the bug data set presented in Section II-C. For RQs 1 and 2, we collected a set of systems intended to be representative and realistic. Our selection criteria were:

- **Programming Language:** We selected ROS systems primarily comprised of C++ code. For systems with a few Python components we manually created the models.
- **Availability of a Simulator:** Our methodology for RQ2 requires that we can execute the system in simulation.
- **Popularity:** We focused on systems that were highly starred on GitHub as representative of the target audience for ROSDiscover.

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

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

We used the data set from Malavolta et al. [19] as a basis for the system selection. The resulting systems are: AutoRally [27], Autoware [28], Fetch [29], Husky, and TurtleBot [30]. Table II provides an overview indicating lines of code for each system, and the number of misconfiguration bugs we considered for RQ3.

A. RQ1 – Component Model Recovery

Methodology: As described in Section III-A, component interfaces cannot always be fully recovered by `ROSDiscover`. Thus, to measure the accuracy of the component model recovery we measure the percentage of API calls that have at least one unknown argument (i.e., where the analysis returns $\top$). This metric is can over-approximate the accuracy by failing to distinguish dynamically loaded source code that cannot be analyzed statically. Nodes with unrecoverable API calls typically require handwritten models for the recovered architecture to be sufficiently accurate. Because understanding and modeling a node is not always proprotional to the number of API calls they include, we also measured the percentage of nodes with at least one "unknown" call as an additional heuristic for modeling effort. We evaluate these metrics on the five systems shown in Table II.

Results: Overall, `ROSDiscover` can recover all information for $87.37\,\%$ of the API calls across the systems we analyzed, as summarized in Table III. Most of the nodes within these systems can be fully recovered. Only $15.19\,\%$ of the nodes contain API calls with unknown arguments, requiring (partially) handwritten models for these nodes.

Listing 1 shows an example of an API call that our approach cannot recover because of the limitations of our symbolic execution. A for-loop iterates over a member variable that changes during execution; the body of the loop contains a subscribe call. The topic name to which the node subscribes depends on the content of the variable. Cases like this are rare in ROS system [4]. To further reduce the specification effort, `ROSDiscover` can provide partial models (i.e., component models containing $\top$ among recovered information), indicating where in the source it failed to recover all information.

Finally, we note that these numbers exclude 29 nodes for which the static analysis crashed. The crashes were caused by Clang incompatibility issues and interface problems between CUDA and Clang. However, these are only a limitation of the implementation, not the general approach.

In summary, these results indicate that the *Component Model Recovery* has high accuracy ($> 85\,\%$) for real systems.

System	API Calls	% of Unknown API Calls	Nodes	% of Nodes with Unknown API Calls
AutoRally	75	13.33	25	16.00
Autoware	882	14.51	209	32.06
Fetch	103	1.94	93	1.08
Husky	223	2.69	105	2.86
TurtleBot	130	14.62	104	1.92
All	1306	12.63	507	15.19

TABLE III: Accuracy of static node recovery per system. Unknown API calls are architecturally relevant ROS API calls for which the static recovery cannot resolve all arguments. "All" describes the union of all nodes, hence includes reused nodes only once.

```
for (auto& mapIt : chassisCommands_)
{
  std::string topic = mapIt.first+"/chassisCommand";
  ros::Subscriber sub = nh.subscribe(topic, 1,
    &AutoRallyChassis::chassisCommandCallback, this);
  chassisCommandSub_[mapIt.first] = sub;
}
```

Listing 1: Example of an API call that is too dynamic to recover using static analysis. `chassisCommands_` is the map of the most recent command from each commander. Source: https://github.com/AutoRally/autorally/blob/c2692f2/autorally_core/src/autorally_chassis/AutoRallyChassis.cpp#L94-L100

B. RQ2 – System Architecture Recovery

Methodology: Measuring the accuracy of static architectural recovery requires a ground truth architecture to compare to the recovered architecture. However, our systems do not provide a reliable, up-to-date ground truth for their architecture (motivating `ROSDiscover` in the first place!). We therefore aimed to run full configurations of the robots and observe their run time architectures. We used the same launch files used for recovery to launch the robot systems, and then observed the architecture dynamically by analyzing the directory of launched nodes, and observing sent messages and topics. Where appropriate we ran the robots through a mission and made multiple observations, merging the observed architectures to have a more complete picture of any changes that occurred over the execution lifetime. We ended up being able to observe only a subset of the systems: AutoRally, Husky, and TurtleBot; we excluded Fetch, because nodes that were different to the other systems examined are written in Python (e.g., ”moveit” and ”demo” leaving not much additional code for `ROSDiscover` to analyze, and Autoware, because all Docker images provided by the developers resulted in some nodes crashing during the execution of a typical scenario.

We use two metrics to compare the recovered architecture with the observed architecture. *Over-approximation* measures the percentage of elements (nodes, connectors) in the recovered architecture that do not appear in the observed architecture. This would occur if the observations did not cover the same code that was covered by the static analysis. *Under-approximation* measures the reverse – the degree to which static recovery did not capture as much as was observed. This is more serious than over-approximation because it would indicate that our analysis has poor coverage of the architecture. We did not use any graph differencing algorithms that might, for example, take into account node or topic renames (i.e., nodes or connections where we recovered names incorrectly) – in our measure, these are marked as deviations. Therefore, our measure is a worst-case comparison that nonetheless gives some indication of accuracy.

We use the fraction of nodes requiring handwritten models that are in the core library as a proxy for modeling effort.

System	observed	recovered	% over approx.	% under approx.
AutoRally	13	13	0.00	0.00
Husky	19	17	5.26	15.79
TurtleBot	12	13	16.67	8.33
All	44	43	6.82	9.09

TABLE IV: Accuracy of static recovery per system. Observed and recovered contains publishers, subscribers, service providers, and action servers. Our systems do not use action clients. "All" contains all configurations, with reused nodes showing up multiple times.

Kind of Arch. El.	obs.	rec.	% over approx.	% under approx.
Providers	8	8	0.00	0.00
Publishers	20	20	10.00	10.00
Subscribers	16	15	6.25	12.50

TABLE V: The data from Table IV per architectural element.

Results: Table IV summarizes results. `ROSDiscover` under-approximates system architectures to 9.09 %, compared to the dynamic analysis. This means that 9.09 % of the architectural elements (i.e., nodes, publishers, subscribers, service providers, action clients, and action servers) that are in the observed architecture are not in the recovered architecture. The overall over-approximation rate is 6.82 %. Over-approximations come mainly from the handwritten models reporting topics that are not used at run time; under-approximations come from plugins and linked libraries loaded at run time that the static analysis does not reach.

`ROSDiscover` does not perform equally well on all architectural elements, as Table V shows. This is usually caused by conditionally launching nodes depending on parameters that cannot be recovered statically. Overall, `ROSDiscover` misses 12.5 % of the subscribe relationships, and 10 % of the publish relationships in recovered nodes. This usually results from the inaccuracies in the component models. For the recovered nodes we have 100 % accuracy for service providers. The studied systems did not have any action clients.

We wrote eight nodes by hand for each of AutoRally, Husky, and TurtleBot. Due to node reuse, these numbers overlap. For all systems, a total number of 15 nodes required handwritten models. Ten of these are nodes are associated with the Gazebo simulator, which is a complex third party library with its own extension mechanism. Hence, they are out of scope of a static recovery approach. Four node models are reusable because they are part of the ROS core library. Moreover, our handwritten nodes are generally accurate (11.30 % over-approximated and 5.02 % under-approximated across all system; Publishers are under-approximated in 6.02 % of the cases, subscribers in 6.25 %, service providers in 3.74 %). Although we are not active developers on the corresponding ROS packages, we were generally able to create accurate models.

Overall, these results suggest that the *System Architecture Composition* has high accuracy (90 %) while keeping the specification effort tractable with 3.67 non-reusable node models per system and 28.78 lines of code per model on average.

C. RQ3 – Bug Detection

Methodology: To measure `ROSDiscover`'s ability to detect architecture misconfiguration bugs, we began from the data set from Section II-C to construct a corpus of bugs. We filtered the data set to focus on systems primarily written in C++ and bugs that relate to topics, services, and actions, since this is the class of architectural elements `ROSDiscover` was designed to recover. For example, we discarded misconfigurations of component parameters. This left 19 misconfiguration bugs that `ROSDiscover` could potentially catch (Table II).

Recall from Section III-C that to reduce false positives, we flag interfaces as potentially misconfigured only if another node is trying to use that interface (i.e., the interface cannot just be dangling). In some of systems, we could not find such nodes, even when it was fixed. For example, the topic `pose_estimate` for `gps_imu` was incorrectly remapped to as `pose_estimate_new` in the buggy version of autorally-01. However, we could not find a subscriber to `pose_estimate` in any configuration in the container (e.g., perhaps because a third party outside the system is expected to subscribe to it). To mimic potential use of the buggy topic in such cases, we introduced a node that subscribes to it (`pose_estimate`, in our example).

To evaluate success, we ran bug detection on the buggy system and checked that the misconfiguration in question was reported (we also confirmed that the error disappeared in the fixed system). To approximate false positives, we assessed the number of errors produced by bug finding on the (otherwise unmodified) systems evaluated in Section IV-B. We assume these architectures to be correct and treat any reported errors as a false positive, as a potential overapproximate measure.

Results: Table VI shows results. We detect eight bugs outright. Further, eight bugs are detectable in principle, but `ROSDiscover` failed to recover the architecture fully. Of these, our static analysis did not work on three (autorally-02, autorally-05, and husky-01) due to implementation limitations; two (autoware-03, husky-03) were afflicted by complicated control flows like those in Listing 1; and two (autoware-05, autoware-09) appeared to be in parts of the software that were not intended to be built. We marked three misconfiguration bugs as undetectable by our approach because they either involve multiple robot configurations, which we do not handle yet (autoware-04), or the bugs involved the removal of dangling interfaces that were never meant to be used (autoware-02, autoware-10). In summary, a rule-based approach on run-time architecture is applicable to architecture misconfiguration bugs in our dataset, and `ROSDiscover` can detect 8 of 19.

Finally, we consider false positives, a key concern in static analysis usability. Because the misconfiguration bugs are reproduced as partial architectures, we instead counted the number of false positives in the complete configurations from RQ2, because these configurations should be complete. For the AutoRally system, the rules result in eight false positives. Husky has five, while TurtleBot has two.

Overall, the results show that `ROSDiscover` finds 42 %

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

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

of the bugs from our real-world data set while keeping the absolute number of false positives per system tractable (< 10).

D. Threats to Validity

Regarding potential threats to *External Validity*, note that our evaluation has only been conducted on open source software. We attempted to include a diverse range of systems; Autoware is by far the largest open-source ROS system. However, the results may not generalize to proprietary systems developed in different ways, or with less reuse.

Considering threats to *Internal Validity*, first, the analysis in RQ2 focuses on a portion of each system corresponding to the bug in question. Hence these results only measure the accuracy for these configurations. Second, the dynamic reconstruction of the architecture for RQ2 can only observe a finite set of executions can be tested. Hence, we might miss architectural elements that were not executed. To mitigate this threat, we manually resolved disagreements between the static and dynamic analyses. However, missing ground truth that manifests in neither analysis still poses a threat that cannot be easily addressed for complex systems, since real ground truth data is unavailable.

Regarding *Construct Validity*, first, the evaluation of node-level accuracy for RQ1 is limited to counting API calls containing arguments designated $\top$; this over-approximates accuracy by failing to distinguish information from dynamically linked libraries. Note however that unknown API calls are part of the evaluation of RQ2 that compares static to dynamically covered architectures, which include code unknown to static analysis. Finally, it is not possible to measure practicality without the tool being used in a real development setting. Our metrics were designed to approximate effort, but truly evaluating usability requires evaluation in context.

V. Related Work

Static Analysis & Bug Detection for Robotics: Static analysis has been used to automatically find bugs in robot systems before. The systems Phriky [31], Phys [32], and Physframe [33] use type checking to find inconsistencies in assignments based on physical units or 3D transformations in ROS code. Thanks to accounting for inter-component dataflow via publish-subscribe, they can also be used to identify inconsistencies of message types. They could be combined with `ROSDiscover` since `ROSDiscover` collects different information and finds different kinds of bugs.

Furthermore, Swarmbug [34] finds configuration bugs in robot systems that result from misconfigured algorithmic parameters, causing the system to behave unexpectedly. In contrast to the misconfiguration bugs that `ROSDiscover` can find, these misconfigurations do not result from incorrect composition or connection of components. Hence, they are not *architectural* misconfiguration bugs.

Module View Recovery: Initial approaches to statically recover architectures relied on clustering software parts based on modularity metrics [35]–[43]. More recent approaches for static recovery of architectures use lexical information [44], concerns [45], and interactive support for architects [46], [47]. 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 [47], [48]. However, in contrast to component-connector views, module views cannot be used to detect architecture configuration bugs, since they do not capture the interactions and composition of components. To this end run-time architectures are needed.

Dynamic Recovery of Run-Time Architectures: In existing work, run-time architectures have mostly been recovered using dynamic analysis. DiscoTect constructs state machines from event traces to reconstruct run-time architectures [49]. Domain-specific reconstructions include Remote Procedure Call (RPC) connectors of CORBA systems based on execution traces [50] and reconstruction of telecommunication systems using traffic case tracing of execution traces [51]. However, for dynamic recovery a large number of system executions have to be performed, which, especially for robotic systems, is time-intensive and might require access to special hardware. Furthermore, dynamic execution might miss cases in rarely executed software. In contrast, our approach uses static analysis, which only assumes that the software can be compiled.

Static Recovery of Run-Time Architectures: The first techniques for static connector recovery operated on abstract syntax trees to find function calls using tree marching [8] or data flow analysis [9]. There are domain-specific recovery techniques for other domains, such as distributed systems [52], However, they are not designed to recover publish-subscribe architectures, such as ROS architectures.

In contrast, our work focuses only on ROS architectures, which enables us to make assumptions on the definition of components and the usage of connectors. Hence, components can be directly mapped to ROS nodes. Connectors such as publish-subscribe can be inferred from calls to the ROS API.

Existing static recovery of run-time architectures for ROS, such as HAROS [10], [11] and Witte et al. [12] are the most closely related to our work. They are limited to API calls that include literals rather than variables. According to Santos et al. [4] about $75\,\%$ of the topic names in API calls are literals. Even if they find all of them, their accuracy is still significantly below our $86.83\,\%$ API call recovery accuracy and $90\,\%$ system-level accuracy, which we achieve through a combination of handwritten models, variable and parameter resolution, and remap parsing.

Architecture Validation: Architecture validation has been used for compliance checking in many domains [6], [53], [54]. Constraint checking is a common way to validate architectural structure [22], [55]. We build on this work to express and check rules for well-formedness in ROS architectures.

VI. Discussion

In this paper we have shown that run-time architectures for ROS systems can be accurately recovered statically and that they can be used to detect common misconfiguration bugs. Our approach exploits the fact that static analysis only needs to focus on architecturally relevant APIs calls and configuration files to recover run-time architectures. While conducting this research, we have made a number of observations.

Static analysis needs variable resolution and path sensitivity: We provide support for variable resolution in our static analysis, unlike other approaches [10]–[12]. On the other hand, these approaches include path sensitivity analysis for conditional API calls. To maximize accuracy, we believe future work should combine these approaches. In fact, the data that we have collected as part of this study can be used to focus improvements to where they would provide the most benefit.

The approach can be used for complex systems: Systems that heavily rely on dynamically linked libraries and custom plugin mechanism are a challenge for every static analysis. This is because during compile time it is unknown which piece of software will be loaded. Since many ROS packages, such as `move_base` or Gazebo, make heavy use of these dynamic linking mechanism, static analysis alone is not a viable solution for realistic systems. These components are also some of the most reused across ROS projects. In contrast to existing work for ROS [10]–[12], we let developers provide handwritten models for pieces of the system that static analysis cannot handle. This enables the analysis of entire ROS systems, which would be impractical otherwise. Furthermore, since the average time to statically recover the API calls from the source code is only $47\,\mathrm{sec}$ per node across all of our studied systems, this approach does scale to more complex systems having hundreds of thousands lines of code. Hence, our approach makes automated architectural analysis more practical for complex systems than existing approaches. Future work is needed to provide more tailored methods for developers to evolve partially handwritten component models, to specify which plugins and libraries are loaded dynamically to support static analysis where code is available, and to handle component communication via parameters.

Rule selection can tailor the approach to use cases: The accuracy of bug detection strongly depends on the rules used for architectural checking. Different rules may be suitable depending on the properties germane to a particular use case or domain. For example, in a continuous integration pipeline, rules that are optimized for minimizing false positives are more practical to avoid disrupting the development process [23], [24]. However before the release of a product, rules that minimize false negatives (e.g., checking for all dangling subscribers) can be used while taking more time to go through a longer list of potential issues. Specific rule violation instances can be whitelisted if they are identified as false positives to avoid checking them again. Future work can identify which specific ruleset is suited for the relevant use cases. One of the challenges in crafting these rules is the simple fact that we are really trying to identify those interfaces that are critical to component functionality. If ROS provided mechanisms to encourage developers to supply this information, perhaps in the existing package specification, then architectural checking could be much simplified. Machine learning over a large corpus of ROS examples might also be able to further tailor these rules based on common usage patterns.

The approach is generalizable: We believe that the approach to analyze API calls of component-based architectural frameworks to statically recover run-time architectures applies to other ecosystems in which the framework APIs have well-defined architectural semantics, as well as configuration files that define the run-time deployment of the systems. For example, NASA's component framework F' uses XML-based architectural description and configuration and provides an API for using architectural connectors similar to ROS [14]. Furthermore, the microservice framework Kubernetes uses Representational State Transfer (REST) APIs for letting service components communicate with each other. Another aspect of our approach is that it relies on a quasi-static architecture that changes little as the program runs. We suspect that many frameworks (e.g., Spring, Kafka) have a similar characteristic. This opens up future work on applying this approach to other domains, and other frameworks. A small step in this direction would be to extend `ROSDiscover` to support Python, as well as to support ROS 2, which is similar but has certain key architectural differences as compared to ROS 1.

Architectural recovery supports reverse engineering: While in this paper we focused on architecture recovery as a means for detecting bugs, it can be used more generally. For example, it can be used to get up-to-date architectural documentation that can help developers to quickly understand a package they are planning to reuse. Further, new developers joining the team can use the architecture to become familiar with the project by inspecting the interface of components and understanding major connectors. So architectural information can help with the broader task of understanding how to correctly use complex frameworks. Future work is needed to tailor architectural recovery to reverse engineering tasks.

References

[1] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Towards component-based robotics," in *International Conference on Intelligent Robots and Systems (IROS '05)*, IEEE, 2005, pp. 163–168, DOI: 10.1109/IROS.2005.1545523.

[2] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007, DOI: 10.1007/s10514-006-9013-8.

[3] P. Estefo, J. Simmonds, R. Robbes, and J. Fabry, "The Robot Operating System: Package reuse and community dynamics," *Journal of Systems and Software (JSS)*, vol. 151, pp. 226–242, 2019, DOI: https://doi.org/10.1016/j.jss.2019.02.024.

[4] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos, "Mining the usage patterns of ROS primitives," in *International Conference on Intelligent Robots and Systems (IROS '17)*, IEEE, 2017, pp. 3855–3860, DOI: 10.1109/IROS.2017.8206237.

[5] C. Timperley and A. Wasowski, "188 ROS bugs later: Where do we go from here?" In *ROSCon Macau 2019*, Open Robotics, Oct. 2019, DOI: 10.36288/ROSCon2019-900898.

[6] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *Working Conference on Software Architecture (WICSA '07)*, IEEE, 2007, pp. 12–12, DOI: 10.1109/WICSA.2007.1.

[7] I. Malavolta, G. A. Lewis, B. Schmerl, P. Lago, and D. Garlan, "Mining guidelines for architecting robotics software," *Journal of Systems and Software (JSS)*, vol. 178, p. 110969, 2021, DOI: https://doi.org/10.1016/j.jss.2021.110969.

[8] D. Harris, H. Reubenstein, and A. Yeh, "Recognizers for extracting architectural features from source code," in *Working Conference on Reverse Engineering (WCRE '95)*, IEEE, 1995, pp. 252–261, DOI: 10.1109/WCRE.1995.514713.

[9] R. Fiutem, P. Tonella, G. Anteniol, and E. Merlo, "A cliche-based environment to support architectural reverse engineering," in *Working Conference on Reverse Engineering (WCRE '96)*, IEEE, 1996, pp. 277–286, DOI: 10.1109/WCRE.1996.558936.

[10] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ros repositories," in *International Conference on Intelligent Robots and Systems (IROS '16)*, IEEE, 2016, pp. 4491–4496, DOI: 10.1109/IROS.2016.7759661.

[11] A. Santos, A. Cunha, and N. Macedo, "Static-Time Extraction and Analysis of the ROS Computation Graph," in *International Conference on Robotic Computing (IRC '19)*, IEEE, 2019, pp. 62–69, DOI: 10.1109/IRC.2019.00018.

[12] T. Witte and M. Tichy, "Checking Consistency of Robot Software Architectures in ROS," in *International Workshop on Robotics Software Engineering (RoSE '18)*, IEEE, 2018, pp. 1–8, [Online]. Available: https://ieeexplore.ieee.org/document/8445812.

[13] S. Kolak, A. Afzal, C. Le Goues, M. Hilton, and C. S. Timperley, "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, 2020, pp. 430–440, DOI: 10.1109/ICSME46990.2020.00048.

[14] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, "F Prime: An Open-Source Framework for Small-Scale Flight Software Systems," in *Small Satellite Conference*, 2018, [Online]. Available: https://digitalcommons.usu.edu/smallsat/2018/all2018/328/.

[15] N. Alshuqayran, N. Ali, and R. Evans, "Towards Micro Service Architecture Recovery: An Empirical Study," in *International Conference on Software Architecture (ICSA '18)*, IEEE, 2018, pp. 47–4709, DOI: 10.1109/ICSA.2018.00014.

[16] *Update about the october 4th outage*, Accessed 11-Oct-2021, [Online]. Available: https://engineering.fb.com/2021/10/04/networking-traffic/outage/.

[17] A. Afzal, C. Le Goues, M. Hilton, and C. S. Timperley, "A Study on Challenges of Testing Robotic Systems," in *International Conference on Software Testing, Validation and Verification (ICST '20)*, IEEE, 2020, pp. 96–107, DOI: 10.1109/ICST46399.2020.00020.

[18] A. Afzal, D. S. Katz, C. Le Goues, and C. S. Timperley, "Simulation for Robotics Test Automation: Developer Perspectives," in *Conference on Software Testing, Verification and Validation (ICST '21)*, IEEE, 2021, pp. 263–274, DOI: 10.1109/ICST49551.2021.00036.

[19] I. Malavolta, G. Lewis, B. Schmerl, P. Lago, and D. Garlan, "How Do You Architect Your Robots? State of the Practice and Guidelines for ROS-Based Systems," in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '20)*, ACM, 2020, pp. 31–40, DOI: 10.1145/3377813.3381358.

[20] G. D. Abowd, R. Allen, and D. Garlan, "Formalizing Style to Understand Descriptions of Software Architecture," 4, vol. 4, ACM, Oct. 1995, pp. 319–364, DOI: 10.1145/226241.226244.

[21] B. Schmerl and D. Garlan, "AcmeStudio: Supporting Style-Centered Architecture Development (Research Demonstration)," in *International Conference on Software Engineering (ICSE'04)*, 23-28 May 2004, pp. 704–705, DOI: 10.1109/ICSE.2004.1317497.

[22] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems," pp. 47–68, 2000, [Online]. Available: https://www.cs.cmu.edu/afs/cs/project/able/ftp/acme-fcbs/acme-fcbs.pdf.

[23] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *International Conference on Software Engineering (ICSE '13)*, IEEE, 2013, pp. 672–681, DOI: 10.1109/ICSE.2013.6606613.

[24] M. Christakis and C. Bird, "What Developers Want and Need from Program Analysis: An Empirical Study," in *International Conference on Automated Software Engineering (ASE '16)*, ACM, 2016, pp. 332–343, DOI: 10.1145/2970276.2970347.

[25] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Commun. ACM*, vol. 7, no. 3, pp. 171–176, Mar. 1964, DOI: 10.1145/363958.363994.

[26] G. V. Bard, "Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric," in *Australasian Symposium on ACSW Frontiers (ACSW'07)*, Australian Computer Society, Inc., 2007, pp. 117–124, [Online]. Available: https://eprint.iacr.org/2006/364.pdf.

[27] B. Goldfain, P. Drews, C. You, *et al.*, "AutoRally: An Open Platform for Aggressive Autonomous Driving," *IEEE Control Systems Magazine*, vol. 39, no. 1, pp. 26–55, 2019, DOI: 10.1109/MCS.2018.2876958.

[28] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An Open Approach to Autonomous Vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015, DOI: 10.1109/MM.2015.133.

[29] M. Wise, M. Ferguson, D. King, E. Diehr, and D. Dymesich, "Fetch and freight: Standard platforms for service robot applications," in *Workshop on autonomous mobile service robots*, 2016, [Online]. Available: http://docs.fetch3staging.wpengine.com/FetchAndFreight2016.pdf.

[30] D. Singh, E. Trivedi, Y. Sharma, and V. Niranjan, "TurtleBot: Design and Hardware Component Selection," in *International Conference on Computing, Power and Communication Technologies (GUCON '18)*, IEEE, 2018, pp. 805–809, DOI: 10.1109/GUCON.2018.8675050.

[31] J.-P. Ore, C. Detweiler, and S. Elbaum, "Lightweight Detection of Physical Unit Inconsistencies without Program Annotations," in *SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*, ACM, 2017, pp. 341–351, DOI: 10.1145/3092703.3092722.

[32] S. Kate, J.-P. Ore, X. Zhang, S. Elbaum, and Z. Xu, "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, 2018, pp. 563–573, DOI: 10.1145/3236024.3236035.

[33] S. Kate, M. Chinn, H. Choi, X. Zhang, and S. Elbaum, "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, 2021, pp. 45–56, DOI: 10.1145/3468264.3468608.

[34] C. Jung, A. Ahad, J. Jung, S. Elbaum, and Y. Kwon, "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, 2021, pp. 868–880, DOI: 10.1145/3468264.3468601.

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

[36] S. Patel, W. Chu, and R. Baxter, "A Measure for Composite Module Cohesion," in *International Conference on Software Engineering (ICSE '92)*, ACM, 1992, pp. 38–48, DOI: 10.1145/143062.143086.

[37] L. Belady and C. Evangelisti, “System partitioning and its measure,” *Journal of Systems and Software (JSS)*, vol. 2, no. 1, pp. 23–29, 1981, DOI: 10.1016/0164-1212(81)90043-1.

[38] D. Hutchens and V. Basili, “System Structure Analysis: Clustering with Data Bindings,” *Transactions on Software Engineering (TSE)*, vol. SE-11, no. 8, pp. 749–757, 1985, DOI: 10.1109/TSE.1985.232524.

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

[40] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, “Bunch: A clustering tool for the recovery and maintenance of software system structures,” in *International Conference on Software Maintenance (ICSM ’99)*, IEEE, 1999, pp. 50–59, DOI: 10.1109/ICSM.1999.792498.

[41] O. Maqbool and H. Babri, “The weighted combined algorithm: A linkage algorithm for software clustering,” in *European Conference on Software Maintenance and Reengineering (CSMR ’04)*, IEEE, 2004, pp. 15–24, DOI: 10.1109/CSMR.2004.1281402.

[42] O. Maqbool and H. Babri, “Hierarchical Clustering for Software Architecture Recovery,” *Transactions on Software Engineering (TSE)*, vol. 33, no. 11, pp. 759–780, 2007, DOI: 10.1109/TSE.2007.70732.

[43] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik, “LIMBO: Scalable Clustering of Categorical Data,” in *International Conference on Extending Database Technology (EDBT ’04) - Advances in Database Technology*, Springer, 2004, pp. 123–146, DOI: 10.1007/978-3-540-24741-8_9.

[44] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, “Investigating the use of lexical information for software system clustering,” in *European Conference on Software Maintenance and Reengineering (CSMR ’11)*, IEEE, 2011, pp. 35–44, DOI: 10.1109/CSMR.2011.8.

[45] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, “Enhancing architectural recovery using concerns,” in *International Conference on Automated Software Engineering (ASE ’11)*, IEEE, 2011, pp. 552–555, DOI: 10.1109/ASE.2011.6100123.

[46] L. Chouambe, B. Klatt, and K. Krogmann, “Reverse Engineering Software-Models of Component-Based Systems,” in *European Conference on Software Maintenance and Reengineering (CSMR ’08)*, IEEE, 2008, pp. 93–102, DOI: 10.1109/CSMR.2008.4493304.

[47] Z. T. Sinkala and S. Herold, “InMap: Automated Interactive Code-to-Architecture Mapping Recommendations,” in *International Conference on Software Architecture (ICSA ’21)*, IEEE, 2021, pp. 173–183, DOI: 10.1109/ICSA51549.2021.00024.

[48] D. R. Harris, H. B. Reubenstein, and A. S. Yeh, “Reverse Engineering to the Architectural Level,” in *International Conference on Software Engineering (ICSE ’95)*, IEEE, 1995, pp. 186–186, DOI: 10.1145/225014.225032.

[49] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, “Discovering Architectures from Running Systems,” *Transactions on Software Engineering (TSE)*, vol. 32, no. 7, Jul. 2006, DOI: 10.1109/TSE.2006.66.

[50] J. Moe and D. A. Carr, “Understanding distributed systems via execution trace data,” in *International Workshop on Program Comprehension (IWPC ’01)*, IEEE, 2001, pp. 60–67, DOI: 10.1109/WPC.2001.921714.

[51] A. Marburger and D. Herzberg, “E-CARES research project: Understanding complex legacy telecommunication systems,” in *European Conference on Software Maintenance and Reengineerin (CSMR ’01)*, IEEE, 2001, pp. 139–147, DOI: 10.1109/CSMR.2001.914978.

[52] N. C. Mendonça and J. Kramer, “An approach for recovering distributed system architectures,” *Automated Software Engineering*, vol. 8, no. 3, pp. 311–354, Aug. 2001, DOI: 10.1023/A:1011217720860.

[53] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Mendonça, “Static Architecture-Conformance Checking: An Illustrative Overview,” *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010, DOI: 10.1109/MS.2009.117.

[54] A. Shokri, J. C. S. Santos, and M. Mirakhorli, “ArCode: Facilitating the Use of Application Frameworks to Implement Tactics and Patterns,” in *International Conference on Software Architecture (ICSA ’21)*, IEEE, 2021, pp. 138–149, DOI: 10.1109/ICSA51549.2021.00021.

[55] D. Hou and H. Hoover, “Using SCL to specify and check design intent in source code,” *Transactions on Software Engineering (TSE)*, vol. 32, no. 6, pp. 404–423, 2006, DOI: 10.1109/TSE.2006.60.