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

Tobias Dürschmid

CMU-S3D-25-112

August 2025

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Claire Le Goues (Co-Chair)
David Garlan (Co-Chair)
Christopher Steven Timperley
Ivano Malavolta (Vrije Universiteit Amsterdam)

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Software Engineering

This research was sponsored in part by the National Science Foundation (award numbers CCF1750116, CCF1918140, CNS2148301, and CCF2403061), the University of California Berkeley Foundation (award number A022598), Northeastern University (award number 50265378050), the University of Michigan (award number SUBK00009959PO3005523666), the Portuguese Science and Technology Foundation (award numbers 1031322 and 1031319), Lockheed Martin Corp (award number MRA19001RPS004), the U.S. Army (award number FA8702-15-D-0002), and the Office of Naval Research (award number N000141712899), and IC 066046 (award number 5003209). The views and conclusions presented in this document reflect those of the author and do not necessarily reflect those of the sponsoring organizations.

Keywords: Software Engineering, Robotics, ROS, Software Architecture, Model-based Analysis, Static Analysis, Dynamic Analysis, Visual Diagrams

Abstract

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

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

In this dissertation, I present an approach to automatically infer behavioral models for components of systems based on the Robot Operating System (ROS), the most popular framework for robotics systems, using a combination of static and dynamic analysis by exploiting assumptions about the usage of the ROS framework Application Programming Interface (API) and behavioral idioms. Static analysis looks for architecturally-relevant API calls that implement message sending, handling received messages, sleeping for a periodic interval, and behavioral idioms that implement state-dependent behavior and state transitions. Based on this information, static analysis infers state machine models of architecturally-relevant component behavior. Due to limitations of static analysis, the resulting models are often partial. To complement statically inferred models, I present an approach to instrument the source locations of known unknowns and dynamically observe their values. Then, resulting models will be translated into the common language TLA+/PlusCal used for model-checking. Furthermore, I present a model-based analysis technique to find architecture-misconfiguration bugs in the resulting TLA+ models. Then, I present a technique that automatically generates visual diagrams from the inferred models. Our human study with practicing roboticists and graduate students finds that these diagrams support the understanding of architecturally-relevant component behavior without slowing down participants. This work is a contribution towards making well-proven and powerful but infrequently used methods of model-based analysis more accessible and economical in practice to make robotics systems more reliable and safe.

Acknowledgments

This dissertation would not have been possible without the support and guidance of my advisors, my collaborators, and my friends. First, I would like to thank my familiy and close friends Anna, Emmy, Helen, Rishi, and 서영 for making my time in graduate school more enjoyable and for their continued support. Second, I would like to hugely thank my advisors, Claire Le Goues and David Garlan for their constructive feedback, time, and mental support throughout my Ph.D and for giving me a second chance after initial struggles in my Ph.D. journey. I would also like to thank Eunsuk Kang who taught me many useful research skills that allowed me to finish this dissertation. Additional thanks go to Bradley Schmerl for his continued feedback throughout the years.

Furthermore, I'd like to thank the other members of my committee, Christopher Timplerley and Ivano Malavolta, for their feedback on this dissertation and their time. I'd also like to thank my other collaborators: Siyan Wu, Mohamed Radalla, Levi Busching, it has been an truely wonderful working with all of you.

Last but not least, thanks to everyone who has given me feedback on my talks and paper drafts, including past and present members of squaresLab and ABLE. Special thanks go to Trenton Tabor for sharing his experience with robots and ROS.

Contents

Part I

Context

1 Introduction

Ensuring that robotics systems operate safely and correctly is an important software engineering challenge. As robots are becoming increasingly integrated in work environments and the daily lives of many people [88, 114, 164, 74], their faults can potentially cause dramatic harm to people [12, 84, 163]. However, ensuring that robotics systems are safe and operate correctly is hindered by their large size and complexity [98, 119, 2, 3]. Many robotics systems are comprised of hundreds of thousands of lines or millions of lines of code [158].

Robotics systems, especially systems written for the Robot Operating System (ROS) [140], 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 [4, 89, 33, 140, 158]. Robots can be comprised of hundreds of software components, each of which can have complex behavior [28, 100, 158]. Many ROS systems are predominantly composed of reusable component implementations created by external developers [97]. In this context, the main challenge is their correct composition [158, 34].

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 [69, 34, 159]. In this dissertation, 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 [98, 78, 2, 3].

Software architects commonly use model-based architecture analysis to ensure the safety and correct composition of components [48, 32, 129, 130, 166, 112, 11, 79]. Model-based analysis is a design-time technique to evaluate whether design options meet desired properties. Systems are modeled as a set of interconnected views, such as *component-connector models* (describing what components are in the system, what ports they have, and how ports are connected between components), *behavioral views* (state machines or activity diagrams describing the dynamic reaction a component can have to receiving messages at its ports), and *deployment views* (mapping component instances to processing units) [40]. Using models of interconnected views of the current system, architects can find inconsistencies or predict the impact of changes on the system's behavior.

However, in practice, due to the complexity of robotics systems, creating models manually is time-consuming and difficult [166, 46, 48]. This motivates work on automated model inference to reduce the modeling effort and make formal analysis more accessible in practice.

Architectural recovery techniques, such as `ROSDiscover` [158], HAROS [144, 142], and the tool by Witte et al. [168], can reconstruct component-connector models [40]. Such reconstructions can be effective in finding some bugs resulting from misspellings of communication channel names (called "topics" in ROS) or connectors that connect ports of different message types. However, they do not reconstruct behavioral models. Without behavioral models, model-based analysis cannot reason about dynamic

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

Existing approaches for inferring behavioral models, such as Perfume [131], use exclusively dynamic analysis to infer state machines from execution traces. However, these purely dynamic approaches cannot guarantee that the relationships they find are causal since they observe only correlations within behavior.

Static analysis approaches can have more confidence in the inferred behavioral relationships being causal. On the other hand, the static analysis is limited by the often very dynamic nature of robotics code, which results in values being unknown to the static analysis.

To combine the advantages of both static and dynamic analysis, I present an approach that first uses static analysis and then fills the known unknowns with targeted dynamic analysis that systematically generates execution scenarios based on the results of static analysis.

In general, inferring behavioral models is undecidable [105]. Even a partial solution is practically challenging, because the analysis needs to infer which subset of arbitrary C++ code gets compiled to be executed as a single component, what subset of this component's code communicates with other components, and under what situations this code for inter-component-communication is reachable.

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

1. Component architectures and behaviors are defined via Application Programming Interface (API) calls that have well-understood architectural semantics [143].

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 [143].

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

After inferring behavioral models, our approach automatically translates them into PlusCal/TLA+, one of the most widely used model checking languages, for the detection of bugs. To further support developers, our approach also automatically generates visual diagrams that combine structural and behavioral aspects of the composition of software components. Our human evaluation shows that participants with diagrams could answer questions about the behavior of ROS components more accurately than participants without diagrams.

1.1 Thesis Statement

My thesis statement is:

Thesis Statement

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

The terms used in this statement have the following definitions:

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

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

Practically useful is characterized by how much the models improve ROS developers' understanding of architecturally-relevant behavior and their ability to find architecture misconfiguration bugs.

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

1.2 Contributions

More concretely, to provide evidence for the thesis statement, we present the following main contributions:

1. **`ROSInfer`**: A **static analysis approach** that infers component behavior models from ROS code by looking for API calls of architecturally-relevant behavior and common idioms to implement common behavioral patterns in the ROS ecosystem. This contribution is presented in Chapter 4. It presents evidence towards the thesis statement by showing that assumptions about framework-specific APIs enable the automatic inference of partial behavioral component models for ROS-based robotics systems.

2. **`ROSInstrument`**: A **dynamic analysis approach** that enhances statically inferred models by designing systematic experiments to observe the behaviors of incomplete models. More details on how this approach is implemented are described in Chapter 5. This contribution provides further evidence towards the thesis statement by demonstrating that framework-specific assumptions can be used to complete partial behavioral component models via dynamic analysis.

3. **`ROSFindBugs`**: A **model-based analysis** of the resulting models that translates them into executable PlusCal/TLA+ models with a set of commonly requested analyses from the robotics domain. This approach is described in Chapter 6. This contribution provides initial evidence of the usefulness of the inferred models by demonstrating how they can be used to automatically find bugs. This contribution evaluates the usefulness of inferred models for the use case of finding bugs and demonstrates that the class of bugs is common via a **novel data set** of real-world architecture misconfiguration bugs in open-source ROS systems. The data set is part of the contributions of this dissertation.

4. **`ROSView`**: An approach to automatically generate **visual diagrams** from inferred models and an end-to-end evaluation of the usefulness of these models in an **empirical study** with roboticists.

1.3 Structure of Dissertation

The remainder of the dissertation is structured as follows:

- Chapter 2 explains the required **background** on modeling and analysis of software architectures, robotics, and ROS that is required to deeply understand the contribution of this dissertation and the most closely **related work**.
- Chapter 3 gives a high-level **overview of the presented approach** to infer component-behavioral models for ROS systems by making assumptions on the framework and ecosystem.
- Chapter 4 describes our **static analysis approach** to infer component behavior models for ROS systems and evaluates the accuracy of this approach to support the thesis statement for static analysis.
- Chapter 5 outlines our work on using **dynamic analysis** to complete the statically inferred models by exploiting assumptions about the ROS framework.
- Chapter 6 demonstrates the usefulness of the inferred models by describing domain-specific **model-based analyses** that analyze properties of interest to roboticists on PlusCal/TLA+ as well as a translation from our inferred models to PlusCal/TLA+. It also presents the **bug data set** of architecture misconfiguration bugs used for evaluation and to demonstrate the existence and variety of architecture misconfiguration bugs.
- Chapter 7 demonstrates the usefulness of the inferred models in an **end-to-end evaluation** of visual diagrams automatically generated from the inferred models in a human study.
- Chapter 8 discusses the strengths, weaknesses, and use cases of the main contributions and includes closing remarks.

2 Background and Related Work

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

2.1 Views in Software Architecture

This section explains the background on architectural views that is needed to understand the differences between module views, component-connector views, and dynamic views, as they differentiate the different perspectives used to reason over architectural composition.

Software architectures are usually represented as a set of views that describe different high-level aspects of the system[80].

2.1.1 Module View

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

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

2.1.2 Component-Connector View

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

It contains execution units known as *components* (i.e., independently deployable run-time units, such as processes, objects, or data stores) and their interaction channels known as *connectors*, such as pipes, publish-subscribe, or call-return.

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

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

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

2.1.3 Behavioral View

The *behavioral view*, also known as the dynamic view, expresses the behavior of the system or its parts. It can describe input-output relationships for components, states, state transitions, and actions. Hence, it shows how the components react to inputs. It is often used to reason about concerns such as deadlock freedom, liveness properties, and safety properties.

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

2.2 Architectural Styles

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

2.2.1 Publish-Subscribe

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

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

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

2.2.2 Call-Return

Call-return is an architectural style with two components, the *caller* and the *callee*. The interaction is initiated by the caller by using the interface defined by the callee to send a request. The callee then processes this request and sends a response back to the caller.

There are two variations of the call-return style: asynchronous call-return and synchronous call-return. In the synchronous call-return style, the caller waits until it receives the response and then continues processing with the requested data. In the asynchronous call-return style, the caller continues executing after sending the request and defines a callback that should be called once the response is received.

2.3 Robotics Systems

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

Static analysis and formal model-based analysis have been used to automatically find bugs in robot systems before [7, 112, 11, 79, 135]. For example, the systems Phriky [132], Phys [93], and Physframe [92] use type checking to find inconsistencies in assignments based on physical units or 3D transformations in ROS code.

Furthermore, Swarmbug [87] 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 small number of mostly co-located source locations 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 focuses on architectural bugs.

2.4 The Robot Operating System (ROS)

ROS is the most popular open-source framework for component-based robotics systems. ROS has been deployed on a diversity of robots, including autonomous vehicles, mobile manipulators, underwater systems, and humanoids[1] in environments ranging from industrial warehouses to the International Space Station [115, 16].

To increase the reusability of the more than 8 000[2] software packages in the ROS ecosystem, ROS uses configuration mechanisms and connectors that are not bound during compile time but rather during run time [52]. These mechanisms include string-based identifiers for topics, services, actions, and parameters, as well as remappings between these identifiers [143]. A detailed description of the architecturally-relevant API calls can be found in Section 2.4.4.

ROS comes in two major versions, ROS 1 and ROS 2. The lifetime of ROS 1 is 2010–2025, with the Noetic release of ROS 1 ceasing support in 2025. The community is migrating to ROS 2, which was first released in 2017. ROS 2 maintains similar features to ROS 1, but aims to support a more diverse ecosystem of robots and robotic domains. In this work, we target ROS 1 in particular, as it is the version of ROS with the most open-source systems and the longest history of existing bugs. Therefore, it is most suited to academic research. Future work can extend this work for ROS 2 and other frameworks.

2.4.1 Modules in ROS

Modules in ROS are called *packages*. A package is a unit of code that includes a `package.xml` and can define numerous components, libraries, and plug-ins.

1 https://robots.ros.org [Date Accessed: 18th August 2021]
2 https://index.ros.org/stats/ [Date Accessed: 12th June 2025]

2.4.2 Components in ROS

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

2.4.3 Connectors in ROS

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

Topics Implement a Publish-Subscribe Style

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

Services Implement a Synchronous Call-Return Style

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

Actions Implement an Asynchronous Call-Return Style

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

2.4.4 The ROS API

The ROS API provides common functionality that implements the mechanisms described above. This section presents an overview of the most important ROS API calls.

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

The subscribe call defines a subscriber port. It creates and returns the newly created Subscriber object. The call specifies which callback should be called when the subscriber receives a message.

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

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

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

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

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

Sleep Calls

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

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

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

OK Call: `ros::ok()`

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

2.5 Model-based Analyses

Software architects commonly use model-based architecture analysis to ensure the safety and correct composition of software components [4, 32, 129, 130, 166, 112, 11, 79]. Model-based analysis is a design-time technique to evaluate whether designs meet desired properties. Systems are modeled as a set of interconnected views, such as behavioral views (e.g., state machines or activity diagrams), and component-connector views [40]. Based on models of the current architecture of the system, software architects can find architectural inconsistencies or model changes to predict their impact on the system's behavior.

There has been a large amount of work on model-based analysis of software architectures based on component-connector models [9, 99, 19, 20, 27, 153, 29, 96] and state machines [103, 68, 64, 8]. Since the models we infer follow the same format, our approach makes analyses like these more accessible to developers by reducing the effort to create the models.

2.6 Inference of Module Views

Most approaches for static recovery of software architectures reconstruct structural views of software modules from the perspective of a developer [146, 136, 23, 77, 50, 117, 121, 120, 10, 44, 59, 38]. 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 [72, 152, 66]. Since module views present the code before compilation, they cannot show the relationships of components during run time [40].

2.7 Inference of Component-Connector Models

`ROSDiscover` [158], HAROS [144, 142], and the tool by Witte et al. [168] can reconstruct component-connector models for robotics systems. Component-connector models describe the types of inputs that a component receives, the types of outputs it produces, and to what other components its input and output ports are connected. However, component-connector models do not contain information about how a component reacts to inputs (e.g., what kind of output it produces in response to an input), whether an output port is triggered sporadically or periodically, and whether the component's behavior is dependent on states. Therefore, component-connector models alone cannot be used to analyze the data flow within a system.

2.8 Inference of Behavioral Models

Behavioral models of components can be inferred using dynamic analysis by observing the component behavior of representative execution traces. For example, DiscoTect [145] and Perfume [131] construct state machines from event traces. Similar approaches also use method invariants [101], Linear Temporal Logic (LTL) property templates [106] to increase the effectiveness. Domain-specific approaches have been proposed for CORBA systems [127] or telecommunication systems [122]. The main limitation of dynamic approaches that are based on observation alone is that they can measure only correlations between inputs and states and outputs and cannot make claims about causal relationships. Additionally, approaches relying on dynamic execution might miss cases in rarely executed software. In contrast to this, our approach analyzes the control and data flow of the source code and therefore has the capabilities to differentiate concurrent behavior that just coincidentally happens after an input or state change from behavior that is control-dependent.

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

3 Overview of the Approach

This chapter outlines the high-level ideas and concepts of the main contributions of this dissertation. Figure 3.1 visualizes the data flow between the main contributions. The following chapters will describe each contribution individually.

3.1 Architecturally-relevant Behavior

Figure 3.2 (a) shows an example of a bug from the Autoware.AI [94] system in which the lattice_trajectory_gen component requires an input to perform its main functionality, although no other component sends this message. Hence, lattice_trajectory_gen waits indefinitely.

Existing approaches that recover only component-connector models, such as `ROSDiscover` [158], cannot find this bug because they cannot infer that the input is *required*, i.e., the component's main functionality depends on it. Therefore, they cannot flag it as a bug, as assuming all inputs are required would result in too many false positives.

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

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

Architecturally-Relevant Component Behavior

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

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

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

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

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

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

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

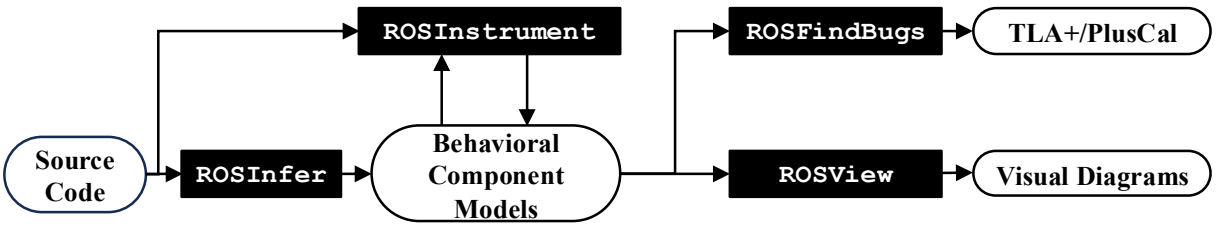


Figure 3.1: Overview of the approach. Boxes represent main contributions. Ovals represent artifacts.

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

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

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

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

Unknown Value $\top$

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

3.2 `ROSInfer`: API-Call-Guided Static Recovery

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

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

We recover reactive behavior by finding control flow from a subscriber callback to a publish call. This establishes one-way causality between receiving a message and sending another message, meaning if a message is sent via the identified publish call, then the component must have received the corresponding message. However, while receiving the input message is required, it is not necessarily sufficient for the component to send the output message, as there could be other conditions.

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

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

We evaluated ROSInfer on 106 components of Autoware,[3] the world's leading open source autonomous driving software, by comparing the recovered behavior with a ground-truth model obtained by manually inspecting the code and creating hand-written models of their actual behavior. If a behavior was not found or a value was not recovered, we traced this false negative back to limitations of the implementation that can be fixed with more engineering effort or limitations of the approach. We find that on our data set, the approach could recover 100 % of periodic behaviors, 84 % of reactive behaviors, 55 % of state variables, and 67 % of state transitions. Detailed methods and results are described in Section 4.2.

3.3 ROSInstrument: Partial-Model-Informed Dynamic Recovery

As the results from the evaluation of our prototype have shown, static analysis still leaves incomplete models in some cases (e.g., due to dynamically-loaded libraries or plug-ins, use of polymorphism, or values loaded at run-time. See Section 4.2 for details). Fortunately, since the models are directly derived from the source code, they could also be used to guide the creation of experiments for dynamic analysis to fill in the unknown values in incomplete models, or to identify representative paths through the system that can be used for profiling. This motivates combining static and automated dynamic analysis to infer behavioral component models that contain more information about the components.

Therefore, we extended ROSInfer with ROSInstrument, a dynamic analysis that automatically deploys components, systematically sends messages to it based on the known state machines to collect timing data or to resolve known unknowns. The details of ROSInstrument and results are described in Chapter 5.

3.4 ROSFindBugs: Model-Checking of Common Properties in Robotics Systems

Combining behavioral component models with component-port-connector models allows for analyses of intra-component-data-flow. Structural models alone do not contain information on how the inputs of a component are used and what is needed for the component to produce an output. Input-output state machine models, like the ones ROSInfer infers, can be used to infer which messages at one component

3 https://www.autoware.org

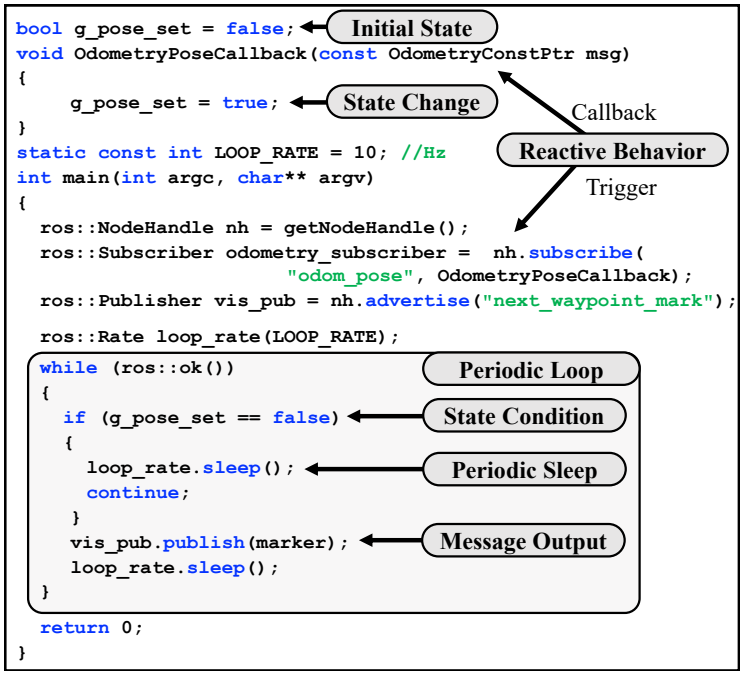
cause messages to be sent from other parts of the system. To check whether the components of a system are composed correctly, properties such as "An input at input port I_1 of component C_a can/must result in an output at output port O_1 of C_b" can be checked via discrete event simulation [29] or logical reasoning [96] (see Chapter 6).

Furthermore, synchronizing the resulting component state machines based on their input/output messages allows for checking arbitrary LTL properties via approaches such as PRISM [104]. 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 [68, 64, 8].

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

3.5 ROSView: Automatic Generation of Visual Diagrams

While existing architectural diagrams can help visualize the connection of software components, commonly used diagram styles are not optimized for visualizing state-based behavioral interactions [40]. On the one hand, high-level perspectives that visualize components and their connections lack the granularity to help developers understand behavioral assumptions. On the other hand, detailed perspectives that visualize a predominant portion of the behavior of a component contain a lot of information that might make it harder to focus on component interactions. Therefore, we present ROSView, an approach that automatically generates visual diagrams that mix structural views of connected components with their architecturally-relevant behavior to provide information that can help developers understand the complex behavior of systems written for ROS. ROSView automatically generates these diagrams from models inferred from code.



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

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

Transitions:

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

pre: $i == odom\_pose$

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

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

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

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

(b) Example model for code shown in Figure 3.2 (a). The first transition handles input at input port *in* and changes the state variable `ready` to true without an output. The second transition triggers periodically with a frequency of 10 Hz if the state variable `ready` is true. Then it sends a message.

Part II

Main Contributions

4 ROSInfer: Static Analysis to Infer Behavioral Component Models

This section describes our approach of static recovery of behavioral component models for ROS systems and the implementation of this approach in our tool called `ROSInfer`. Our approach is based on the observation that reactive, periodic, and state-based behavior of ROS components is often implemented using the API that ROS provides, as shown in Figure 3.2 (a). By looking for the API calls that define callbacks for receiving a message, sending a message, or sleeping for the remaining time of a periodic interval, we aim to recover models of architecturally-relevant behavior that can then be used for model-based analysis of the system.

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

1. An optional **output**. This describes messages being sent from the component. The formalism includes a special element for the non-output.
2. A **trigger**. In component-based systems, there are three types of triggers for architecturally-relevant behavior:
   a) **Reactive triggers:** The behavior was a reaction to a message the component received at a port.
   b) **Periodic triggers:** The behavior is executed periodically. After completion the thread sleeps for the remaining time of the periodic interval so that the message gets sent with a constant frequency[4]
   c) **Component triggers:** The behavior was triggered by a component event (i.e., non-input-related occurrences in the life-cycle of a component, 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 execution of 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.

4.1 Statically Inferring Component Behavior Model

The analysis process and data-flow are shown in Figure 4.1.

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

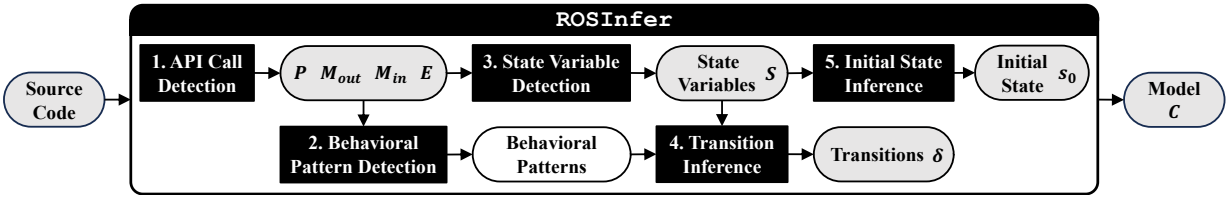


Figure 4.1: Overview of the `ROSInfer` static analysis. Boxes represent analysis steps, gray ovals represent elements of the output model, white ovals represent intermediate results, and arrows indicate data flow.

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

Component Framework API: Inter-component communication for sending and receiving messages happens almost exclusively via API calls that have well-understood architectural semantics [143, 158].

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

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

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

The remainder of the section describes each analysis step.

4.1.1 API Call Detection

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

`ROSInfer` detects the following API calls and behaviors:

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

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

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

To identify control flow entry points ROSInfer looks for callbacks defined as parameters to the ROS API calls NodeHandle::subscribe (see Figure 3.2 (a)), MessageFilter::registerCallback,[5] or NodeHandle::advertiseService.

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

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

ROS offers two filling-time API calls: Rate::sleep, which is called on a rate object (see periodic sleep in Figure 3.2 (a)), and NodeHandle::createTimer, which has a rate object and a callback as arguments. Since the frequency is specified in the constructor of the Rate object, ROSInfer uses constant-folding to infer the frequency's value and denotes it $\top$ if it cannot constant-fold it.

4.1.2 Behavioral Pattern Detection

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

Detecting Reactive Behavior: There are two kinds of reactive behavior: Reacting to receiving a message and reacting to a component event.

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

With respect to component events, ROSInfer detects messages sent in response to "component-started". The pattern looks for publish calls called (transitively) inside the main method and checks if the call can happen in the initial state.

Detecting Periodic Behavior: Periodic publishing behavior is repeated sending of a message of the same type with a constant upper target frequency (note that messages do not necessarily always have to

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

be sent every interval). The pattern to detect periodic behavior is checking for publish calls that happen (transitively) within unbounded loops that (transitively) contain a sleep call. To identify unbounded loops, `ROSInfer` considers loop conditions that are either `true` or `ros::ok()`.

4.1.3 State Variable Detection

The key idea to infer state variables statically is to look for variables in the code that store state information, such as `ready` in Figure 3.2 (a). We use the following heuristics to identify variables that represent component state.

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

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

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

4.1.4 Transition Inference

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

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

4.1.5 Initial Value Inference

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

4.2 Evaluation of Static Analysis

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. Thereby, this section tests the hypothesis that "Assumptions about framework-specific APIs and idioms enable the automatic inference of behavioral component models for ROS-based robotics systems", which is a portion of the overall thesis statement (Section 1.1).

4.2.1 Experimental Setup

To evaluate `ROSInfer` we asked the following research questions:

RQ 1 (Recovery Rate)	**Results in Section 4.2.2**
How high is `ROSInfer`'s recovery rate for real-world ROS systems, i.e., what is the percentage of inferred architecturally-relevant behaviors that can be recovered completely?	

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

RQ 2 (Recall)	**Results in Section 4.2.3**
How high is `ROSInfer`'s recall for real-world ROS systems, i.e. what percentage of architecturally-relevant component behavior can `ROSInfer` infer correctly?	

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

RQ 3 (Precision)	**Results in Section 4.2.4**
How high is `ROSInfer`'s precision for real-world ROS systems, i.e., what percentage of inferred architecturally-relevant component behaviors are true positives?	

Since `ROSInfer` uses heuristics to infer architecturally-relevant behaviors, it can incorrectly classify behaviors as periodic or reactive to a component event or component input, and can include unnecessary

or incorrect state variables or state transitions. To evaluate how many false positives are in the inferred models, i.e., how often ROSInfer infers behaviors that do not exist in the real program, we measured the precision of inferred models compared to a ground truth. This metric measures the degree of soundness of ROSInfer's inference heuristics on real-world ROS systems.

Overview of Evaluation Systems: For all research questions we evaluated ROSInfer on five large real-word open source systems: Autware.AI [94], AutoRally [67], Fetch [167], Husky, and Turtlebot [151].Some components are part of multiple of these systems due to component reuse, leaving 542 unique components in total.

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

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

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

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

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

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

4.2.2 Measuring Recovery Rate (RQ1)

Methodology: As discussed in Section 4.1, ROSInfer denotes values that cannot be statically recovered with $\top$ to indicate unknown values. So the main metric for the recovery rate is how often $\top$ is included in parts of the resulting model.

Table 4.1: Results for RQ1: The trigger types recovery rate is the percentage of inferred publish calls for which `ROSInfer` can infer what kind of trigger causes that behavior (periodic or reactive). For each sub-type of behavior, percentages show how many of that type do not contain unknowns ($\top$) in the inferred modes of a total of 518 components of the five large real-world systems. N is the total number of behaviors of the respective type inferred by `ROSInfer` (all publish calls in the case of trigger types). The All row counts components that are included in multiple systems only once.

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

Table 4.2: Recall and precision of `ROSInfer` based a comparison with 148 manually inferred component models. *TP*, *FP*, and *FN* are the number of true positives, false positives, and false negatives compared to the ground truth models.

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

We ran `ROSInfer` on all 542 components of the five systems Autoware, AutoRally, Fetch, Husky, and Turtlebot. Components that are includes in multiple systems only count once. For 12 components the static analysis crashed due to errors in Clang, 7 timed out after 1 hour, so these components are excluded from the evaluation, leaving 518. For each type of architectural behavior we then calculated the percentage of unknowns included in inferred values (i.e., target frequencies for periodic behavior, triggering events or callbacks for reactive behavior, initial values for state variables, and new values for state transitions). These numbers represent how well `ROSInfer` can infer all parameters of a detected behavior.

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

Trigger Types Recovery Rate (Evaluation Metric)

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

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

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

Results for RQ 1 (Recovery Rate) **See Table 4.1**

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

Results: Detailed quantitative results are shown in Table 4.1.

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

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

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

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

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

4.2.3 Measuring Recall (RQ2)

Methodology: After creating the handwritten models as ground truth (see Section 4.2.1) we executed `ROSInfer` on the source code and compared the results by treating the handwritten models as ground truth. The existence of model elements is compared automatically, while expressions in conditions are compared by humans to judge whether they are logically equivalent. After the quantitative analysis, we then manually inspected each false negative to conduct a qualitative root cause analysis of missed behaviors.

Results for RQ 2 (Recall) **See Table 4.2**

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

Results: Detailed quantitative results are shown in Table 4.2.

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

Cases in which ROSInfer cannot recover state variables include **complicated object logic**, such as whether a list or map is empty (e.g., vscan2image). Figure 4.2 shows an example of this. This requires a deeper understanding of the objects owned by the component that are used to represent its state and are therefore a limitation of the approach. Conditions on object fields can contain implicit dependencies that cannot easily be inferred statically. For example when a subscriber callback initializes the image stored in a state variable whose width and height are checked to be positive numbers in a control condition (`image.width > 0 && image.height > 0`) a human developer can infer that this condition refers to checking whether the initialization in the subscriber callback has been called implying that the component has received the message. This dependency that is implicit due to complex logic within the imagine object cannot be inferred statically.

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

Figure 4.2: Simplified code snippet showing an example from waypoint_clicker in Autoware.AI for which our approach cannot recover the state machine. The analysis would need to model the state of a vector map containing multiple arrays and identify that the assignment in the `cache_point` subscriber callback affects the return value of the `empty` call.

4.2.4 Measuring Precision (RQ3)

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

Results for RQ 3 (Precision) **See Table 4.2**

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

Results: Detailed quantitative results are shown in Table 4.2.

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

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

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

4.2.5 Measuring Execution Time

When running `ROSInfer` on Autoware.AI on a server with 4 Intel(R) Xeon(R) Gold 6240 CPUs (each has 18 cores at 2.60 GHz) with 256 GB RAM, the static analysis took on average 44.93 s. The fully automated analysis of the entire Autoware.AI system took 4.58 h and much shorter for the other systems (Autorally: 18.23 min, Fetch: 31.25 min, Husky: 35.28 min, Turtlebot: 49.40 min). This should demonstrate that the static analysis scales to real-world systems and could integrate well into iterative software development practices.

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

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

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

4.2.6 Lessons Learned about ROS Components

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

1. Many components are designed to process input streams and publish processed outputs like a pipes and filters architecture. These components are stateless and usually produce a single output for each input that they receive.

2. Components that maintain states are often components that start to publish periodically after receiving a set of input messages that are used to initialize the component, such as the example shown in Figure 3.2 (a).

3. Only a few components implement a complex state machine. Most explicit or implicit state variables are booleans and only few components have more than three state variables.

4. While the state machines that model the behavior of the component might be less complex, developers sometimes use more complex language features to express them than would be necessary (see Figure 4.2). This makes the code more extensible and easier to read by human developers, but harder to analyze using static analysis.

4.3 Discussion

4.3.1 Incomplete Models

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

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

Unknown unknowns include missing output behavior (e.g., a publish call was not found due to unavailability of the source code or use of non-ROS-API calls), missing state variables (e.g., due to state implementations without explicit variables), and missing state changes (e.g., due to unidentified state variables or unavailable source code of the state change). Unknown unknowns are more limiting in practice, since it is much harder for users to identify that information is missing from the generated models. Unknown unknowns can be reduced by extending the list of behavioral patterns to look for or by adding the APIs of commonly used libraries, but cannot be fully eliminated.

4.3.2 Coding Style Guidelines

Unlike many open-source ROS systems, most industrially developed projects follow coding style guidelines that narrow down the expected kinds of behaviors by telling developers to implement certain types of code in a certain way. We expect the recall of our approach to benefit from this, because fewer cases of unnecessarily complex versions of simpler code would exist. This effect can become even stronger if coding styles related to specifying architecturally relevant behavior are established, as almost all cases

in which our approach cannot correctly infer architecturally-relevant behavior, the corresponding code could be refactored towards more analyzable code.

For example, coding style guidelines, such as "component states should be explicitly modeled as variables in the code" to avoid the limitation described in Figure 4.2 by replacing `empty()` calls with a state variable, "state variables should be initialized explicitly" to avoid unknown or ambiguous initial states, or "ROS connectors should be used where possible" to avoid over-use of project-specific APIs.

Similarly to how testability became a goal of software design to reduce the effort of ensuring correctness via testing, analyzability of code could become a future design goal of ROS code to support the automatic inference of rich behavioral models for automated formal analysis.

4.4 Conclusions and Implications for the Dissertation

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

5 ROSInstrument: Completion of Behavioral Models using Dynamic Analysis

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

Therefore, I extended ROSInfer with ROSInstrument, a dynamic analysis that automatically instruments the code to log missing values, deploys the components, and observes the values dynamically.

5.1 Motivating Example

ROS developers often structure their code in a way that is harder to analyze statically. For example in the decision_maker_node, Autoware developers decided to store publishers and subscribers in a dictionary rather than independent variables:

```
Pubs["detection_area"] = nh_.advertise<visualization_msgs::Marker>("/state/
    detection_area",1);
Pubs["crossroad_bbox"] = nh_.advertise<jsk_recognition_msgs::BoundingBoxArray>("/
    state/crossroad_bbox", 10);
Pubs["crossroad_marker"] = nh_.advertise<visualization_msgs::MarkerArray>("/state/
    cross_road_marker", 1);
Pubs["stopline_target"] = nh_.advertise<visualization_msgs::Marker>("/state/
    stopline_target",1);
```

The corresponding publish calls are then made like this:

```
Pubs["detection_area"].publish(detection_area_marker);
Pubs["crossroad_bbox"].publish(bbox_array);
Pubs["crossroad_marker"].publish(marker_array);
Pubs["stopline_target"].publish(stopline_target_marker);
```

While the complexity of the resulting program is not larger than it would be if developers had used independent variables rather than a dictionary (detection_area_pub instead of Pubs["detection_area"]), analyzing this program is much harder for static analysis. Static analysis would need to be able to reason about the possible states of individual elements with the dictionary data structure. In the general case, this would be quite complicated as dictionary keys could have other values besides string literals. So static analysis is reaching its practical limits in cases like those.

To still allow us to infer the behavioral component model correctly, we can address these challenges with dynamic analysis. For values that static analysis cannot infer, we can instrument the code to log their values, execute the program, and use the observed values to complete the models.

5.2 Approach

ROSInstrument has three steps: (1) **code instrumentation**, which adds targeted logging statements in the code locations of known unknonws, (2) **component observation**, which deploys the component feeds inputs and observes the component's behavior, and (3) **model inference**, which takes the logged values and completes the statically inferred models with the observed values. Each step is described in the following sections.

The approach of ROSInstrument is based on two main assumptions: (1) Line additions in relative position to the locations of known unknowns based on knowledge about the ROS API and common implementation idioms result in compilable code instrumentation in most cases for real-world ROS code. (2) The dynamic observation of logged values of known unknowns allows the correct inference of model values in many cases for real-world ROS code.

5.2.1 Code Instrumentation

Instrumentation starts from the partial models inferred by ROSInfer, identifies known unknowns ($\top$), and then adds logging statements in the corresponding code locations. To ensure that the logging has minimal impact on the build process of the ROS system, we use the built-in ROS function ROS_ERROR to log values. As this function is defined in the console header file of the ROS library, ROSInstrument adds "`#include <ros/console.h>`" on top of every instrumented file.

The known unknowns inferable include topic names, rates, initial states, and state transitions.

Topic Names: Topic names of subscribers and publishers are defined in the corresponding arguments to the subscribe and advertise calls. If these values cannot be constant-folded by static analysis, ROSInfer denotes them with $\top$ to indicate unknown topic names. For a given partial, statically inferred model, ROSInstrument identifies all publisher / subscriber variables with unknown topic names. It then adds logging statements right after the corresponding subscribe / advertise call is made. For publisher pub_var with unknown topic name, first ROSInfer looks for their assignment by searching for matches of the regular expression "{pub_var}\s+=". Then ROSInstrument adds the following line right after that matching statements:

```
ROS_ERROR("ROSINSTRUMENT ({tag}): " + {pub_var}.getTopic().c_str());
```

This instrumentation exploits knowledge about the ROS API by quering the topic name directly. To also cover cases in which we cannot find the variable assignment statement (e.g., if publishers are created in loops) and to identify if topics change during run time, ROSInstrument also logs publishers' topic names right after each publish call is made.

Rates: Rate frequencies are defined in arguments to the Rate constructor (e.g., `ros::Rate loop_rate(LOOP_RATE); // Try to loop in "LOOP_RATE" [Hz]`), which can result in *top* if constant folding fails. To identify the rate, ROSInstrument instruments the code to log the value of the expected cycle time.

ROSInstrument adds this line:

```
ROS_ERROR("ROSINSTRUMENT ({tag}): %d", {rateVar}.expectedCycleTime().toSec());
```

right after the rate constructor. This instrumentation exploits knowledge about the ROS API by querying the rate cycle time directly.

This will log the duration of the periodic behavior in seconds.

Initial States: In some cases, ROSInfer cannot correctly identify the initial values of state variables, for example, when initialization happens implicitly via the default value of the corresponding data type, or when the initial assignments are in an unexpected code location. To identify which value conceptually corresponds to the initial value, ROSInstrument instruments the code to log every single use of the state variable in expressions with the intention to use the very first one as the initial state. So ROSInstrument searches for all expressions containing the state variable and adds this line right before it: `ROS_ERROR("ROSINSTRUMENT ({tag}[{varName}]): %s", (std::to_string({codeVarName})).c_str());`

State Transitions: Assignments to state variables can result in known values if constant-folding fails. To log the values to which a state variable is assigned in a particular transition, ROSInstrument logs the value right after the assignment using this logging statement: `ROS_ERROR("ROSINSTRUMENT ({tag}[{varName}]): %s", (std::to_string({codeVarName})).c_str());`

5.2.2 Component Observation

After compiling the instrumented components, ROSInstrument takes a user-provided launch file for a representative configuration of the system, launches the system, and then parses the instrumented output. Users can also provide a bag file that simulates external inputs, if needed. The bag file is then replayed in the testing environment.

5.2.3 Model Inference

The final step of the dynamic analysis is to aggregate observations to complete statically inferred component models. ROSInstrument's approach varies based on the different model element types.

Topic Names: Topic names in the model correspond to the observed topic name query result on the publisher / subscriber object in the code. For topic names of a certain publisher / subscriber, a variation in observed values implies that the architecture is non-static and configurable. Since these observations alone cannot guarantee the correct inference of causality, ROSInstrument treats different values as a non-deterministic choice in the model. This means that variation is captured in the model without describing when each value is chosen.

Rates: Rates in the model correspond to the observed expected cycle time query result on the rate object in the code. Analogous to topic names, different values for expected cycle times are modeled as a non-deterministic choice.

Initial States: Initial states in the model correspond to the observed value of the first assignment of a state variable in the code. For each execution, ROSInstrument selects the first observed value of the corresponding state variable as the initial state. If multiple executions have different values, the ROSInstrument models those using non-deterministic choice.

State Transitions: State transitions in the model correspond to the observed value of assignments of state variables in the code. Variation in those observed values is modeled as a non-deterministic choice in the model.

5.3 Evaluation

This section evaluates the effectiveness of ROSInstrument on real-world ROS systems.

The first assumption made by ROSInstrument is that line additions in relative position to the locations of known unknowns, based on knowledge about the ROS API and common implementation idioms, result in compilable code in most cases for real-world ROS code. To evaluate this assumption, we pose the following research question that evaluates whether ROSInstrument's code instrumentation creates sound code for real-world ROS code.

RQ 1 (Compilation Rate) **Results in Section 5.3.1**

How high is ROSInstrument's compilation rate for real-world ROS systems, i.e., what is the percentage of nodes that correctly compile after instrumentation?

The second assumption ROSInstrument made is that the dynamic observation of logged values of known unknowns allows the correct inference of model values in many cases for real-world ROS code. To evaluate this assumption, we pose the second research question that evaluates how effective ROSInstrument is at completing statically inferred models for real-world ROS systems.

RQ 2 (Recovery Rate) **Results in Section 5.3.2**

How high is ROSInstrument's recovery rate for real-world ROS systems, i.e., what is the percentage of model elements that were missing from static analysis and can be correctly completed using dynamic analysis?

Finally, to evaluate the additional resources needed to complete models using ROSInstrument, we pose the third research question that evaluates the time overhead of ROSInstrument.

RQ 3 (Overhead) **Results in Section 5.3.3**

How high is ROSInstrument's overhead, i.e., what is the additional time needed to instrument the code and make observations?

Experimental Setup:

To evaluate the research questions, we ran ROSInstrument on 10 nodes, checked whether the resulting code correctly compiled and manually verified the output of model inference.

We used the same dataset used for the evaluation of ROSInfer (see Section 4.2) and randomly selected nodes for which ROSInfer's output contains known unknowns. Due to the more complicated evaluation setup, we only evaluated on a subset of those partial nodes. The nodes in this evaluation are: waypoint_marker_publisher, kitti_player. catvehicle, obstacle_avoid, watchdog, decision_maker_node, way_planner, adis16470_node, lidar_fake_perception, and scan2image.

5.3.1 Measuring Compilation Rate (RQ1)

We selected 10 nodes with partial models. Out of those, 8 (80.00 %) compiled correctly after instrumentation.

Results for RQ 1 (Compilation Rate)

`ROSInstrument`'s compilation rate on 10 nodes was 80.00 %.

This confirms the assumption that additions in relative position to the locations of known unknowns based on knowledge about the ROS API and common implementation idioms result in compilable code in most cases for real-world ROS code.

5.3.2 Measuring Recovery Rate (RQ2)

Nodes that compiled correctly, but for which we were unable to find a runnable launch file (e.g., `catvehicle`), were excluded from this part of the evaluation.

To measure the recovery rate, we removed instrumented lines that caused syntax errors, but did not fix the underlying issues. To measure recovery rates, those are counted as a miss.

In our evaluation dataset, 13 topics were unknown. `ROSInstrument` could correctly infer 5 (38.46 %) of those.

4 rates were unknown. `ROSInstrument` could correctly infer 1 (25.00 %) of those.

10 initial states were unknown. `ROSInstrument` could correctly infer 1 (10.00 %) of those.

Results for RQ 2 (Recovery Rate)

`ROSInstrument`'s recovery rate on 10 nodes was 38.46 % (of 13) for topics, 25.00 % (of 4) for rates, and 10.00 % (of 10) for initial states.

These results show that the assumption that the dynamic observation of logged the values of known unknowns allows the correct inference of model values for real-world ROS code holds for some, but not most, nodes. As there were no incorrect values, the results from running `ROSInstrument` after `ROSInfer` result in models that are equal to or better than the original `ROSInfer` models. Therefore, they show an improvement.

5.3.3 Measuring Overhead (RQ3)

The time overhead of `ROSInstrument` includes three aspects: (1) *instrumentation time* (i.e., the time it takes to generate code changed from partial models), (2) *compilation time* (i.e., the time it takes to compile the instrumented code), and (3) *execution time* (i.e., the time it takes to execute the instrumented code).

`ROSInstrument`'s instrumentation time is instant.

The compilation time depends on the instrumented project / nodes. For Autoware, the compilation time is 6 min 31 s with instrumentation and 6 min 30 s without instrumentation. This negligible difference implies that instrumentation does not meaningfully impact compilation time.

The execution time consists of the *setup time* (i.e., the time to launch the nodes in the configuration) and *observation time* (i.e, the time it takes to make observations). The average *setup time* was 8.2 s. In our experiments, we configured `ROSInstrument` to observe for a constant amount of time (1 min). This brings the per-node execution time per node to 68.2 s.

Results for RQ 3 (Overhead)

ROSInstrument's overhead includes the per-system compilation time (6 min 31 s for autoware with negligible difference to non-instrumented code) and the per-node execution time (68.2 s).

These results show that the ROSInstrument's overhead is not significantly larger than the overhead of ROSInfer (which was 44.93 s per node).

5.4 Related Work

Existing work from non-ROS software domains [30, 31, 81] observes the behavior of a running system to extract behavioral models. Since existing work is not designed for the ROS framework, it cannot infer topic names or frequencies of periodic publishing behavior. In contrast, ROSInstrument exploits assumptions about framework-specific APIs of the ROS API to infer periodic frequencies and topic names that existing approaches are unable to do, because they are not designed specifically for the ROS framework. Furthermore, ROSInstrument uses statically inferred partial models to guide the instrumentation of source code to fill in the gaps in the partial models rather than using a black-box approach to generate a model only from dynamic observations. Thereby, models inferred using ROSInstrument benefit from the information captured in static analysis of periodic behavior and reactive behavior and thereby combine advantages from dynamic and static analysis to infer behavioral models with higher confidence than models that find correlations in execution traces to guess whether behavior is reactive, periodic, and/or state-based.

5.5 Discussion

This section discusses practical limitations of ROSInstrument.

Unsound Code: In some cases, ROSInstrument creates code that does not compile due to type errors or statements being placed in incorrect locations. For example, in the kitty_player node, ROSInfer detects cv_image03 as a state variable, which is of type cv::Mat. So the generated log for initial values does not compile: ROS_ERROR("ROSINSTRUMENT (INITIAL[cv_image03]): %s", (std::to_string(cv_image03)).c_str()); In these cases, developers can manually fix or remove the incorrect statements.
Alternatively, this challenge could potentially be addressed by investing additional engineering effort into improving the correctness of code instrumentation.

Coverage Limitations: In all cases in which ROSInstrument could not correctly recover a known unknown, the reason was that the particular line was not covered during the execution of the program. Manual inspection came to the conclusion that if the lines were covered, ROSInstrument would likely have correctly recovered the correct values. This suggests that the effectiveness of the approach, for typical ROS systems, mainly depends on how high the coverage of instrumented lines is. In some cases, the lines were not covered because the configuration requires additional hardware. For example, the adis16470_node requires a device to be connected to start the node, which is hard to provide in a testing environment. This could be overcome with hardware-in-the-loop simulators [109]. Using this technique, external hardware could be part of the simulation and therefore result in higher coverage. Furthermore, fuzzing techniques [116] could be used to generate inputs that are likely to bring components into the

state in which they should be observed.

5.6 Conclusions and Implications for the Dissertation

This chapter presented `ROSInstrument`, an approach to complete the partial models via dynamic analysis. The results from this chapter demonstrate that dynamic analysis that exploits assumptions about framework-specific APIs can slightly improve statically inferred behavioral component models for ROS-based robotics systems.

6 ROSFindBugs: Model-based Analyses for Automated Bug Finding

After showing that behavioral component models can be inferred from code, this chapter shows that the models can be used to automatically find bugs. The chapter presents an `ROSFindBugs`, an approach to translate models inferred by `ROSInfer` and `ROSInstrument` into PlusCal/TLA+ and check them to find three real-world bugs. Finally, we show that the types of bugs that can be found with `ROSFindBugs` (i.e., behavioral architecture composition bugs) are commonly found in ROS systems by presenting a novel data set of behavioral architecture composition bugs.

6.1 Generation of PlusCal/TLA+ Models

Based on the inferred state machine models (see Chapter 4) and dynamic analysis (see Chapter 5) we automatically generate analyzable PlusCal/TLA+ models, such as the ones shown in Listing A.1, Listing A.2, and Listing A.3.

Components are modeled as fair processes with each transition specified as a labeled action that contains an if-statement with with the transition's pre-conditions. The transaction's post-conditions are then specified in the true branch of the if-statement.

See this example of a PlusCal model of a component with two input ports and one output. After the component has received a message in the first input port, it then forwards all messages of the first port to the output port:

```
fair process component_name ∈ Component_name
variables
  msg ∈ Data;
  ready_ = FALSE;
begin

sending_transition_name:
  if input_queue ≠ <> then \* checking message arrival at the input port *\
    msg := Head(input_queue);
    input_queue := Tail(input_queue);
    if (ready_) then \* state condition *\
      output_queue := output_queue ⊕ msg; \* sending a message by adding it to the
     outpute queue *\
    end if;
  end if;

state_changing_transition_name:
  if input_queue_2 ≠ <> then \* checking message arrival at the input port *\
    msg := Head(input_queue_2);
```

```
    input_queue := Tail(input_queue_2);
    ready_ := TRUE \* state change *\
  end if;
end process;
```

We specify state variables (e.g., `ready_`) as process variables maintained by the corresponding component to avoid name conflicts with state variables of other components and to fit the mental model of state being owned by the corresponding component. We model input queues and output queues of ports as lists with their corresponding queue size as length and declare them as variables of the algorithm to allow the topic process to access and change the input / output queues. Topics are modeled as processes that take elements from the output queues of publishers and then add them to the input queues of subscribers.

For example:

```
fair process image_topic ∈ Image_topic
begin
  Write:
  if output_queue ≠ <> then
    msg := Head(output_queue);
    output_queue := Tail(output_queue);
    subscriber_1_input_queue := subscriber_1_input_queue ⊕ msg;
    subscriber_2_input_queue := subscriber_2_input_queue ⊕ msg;
    subscriber_4_input_queue := subscriber_3_input_queue ⊕ msg;
  end if;
end process;
```

In more detail, the TLA+ generation follows this grammar:

```
--------------------------- MODULE <system_name> ----------------------------
EXTENDS Sequences, Integers, TLC, FiniteSets
CONSTANTS <topic_names> <component_names> Data, NULL, MaxQueue

ASSUME NULL ∉ Data

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

(*--fair algorithm polling
variables
  <topics_init>
  <inports_init>
  <initial_states>

define
  TypeInvariant == <variables_types>
```

```
  <properties>
end define;

<components>
<topics>
```

```
<topic_names> ⟶ ϵ | <topic_name>, <topic_names>
```

```
<component_names> ⟶ ϵ | <component_name>, <component_names>
```

```
<components> ⟶ ϵ | <component> <components>
```

```
<topics> ⟶ ϵ | <topic> <topics>
```

```
<system_name> ⟶ str
```

```
<topic_name> ⟶ str
```

```
<component_name> ⟶ str
```

The individual parts of the specification are described in the following.

6.1.1 Components

Since components can run in parallel in separate threads, we specify them as separate processes in PlusCal. The processes are marked as `fair` to simulate a scheduling mechanism that allows all components to eventually make progress. The component type name in the TLA+ model corresponds to the node type name, while the instance name corresponds to the name of the component described in the launch file. We then generate a list of transactions, which can be reactive or periodic. This is the resulting grammar:

```
<component> ⟶
fair process <component_instance_name> ∈ <component_type_name>
<out_variables>
begin
  <transitions>
end process;
```

```
<transitions> ⟶ ϵ | <transition> <transitions>
```

```
<transition> ⟶ <reactive_transition> | <periodic_transition>
```

```
<component_instance_name> ⟶ str
```

```
<component_type_name> ⟶ str
```

6.1.2 Reactive Transitions

A transition that is a reactive behavior to a message is modeled as a TLA action based on this grammar:

```
<reactive_transition> ⟶
<in_port_name>_action:
  if <in_port_name> ≠ <> then
    msg := Head(<in_port_name>);
    <in_port_name> := Tail(<in_port_name>);
    <behavior>
  end if;
```

```
<in_port_name> ⟶ str
```

To avoid any name conflicts between actions within the component, the name of the action is based on the name of the input port. First, the action checks whether the reactive behavior should be triggered via a pre-condition check for the input queue being empty (`if <in_port_name> ≠ <> then`). If the queue of the input port is not empty, then the message gets removed from the queue (`msg := Head(<in_port_name>);`) and the new value of the queue is assigned to the tail of the queue (`<in_port_name> := Tail(<in_port_name>);`). Finally, the behavior triggered by the message is added, which includes a condition, state changes, and message outputs.

6.1.3 Periodic Transitions

The PlusCal specification of periodic transitions is much simpler and includes only a label generated from the frequency and the specification of the behavior.

```
<periodic_transition> ⟶
  <frequency>_Hz_action:
    <behavior>
```

```
<frequency> ⟶ float
```

6.1.4 Behavior

Behavior (reactive or periodic) that has been triggered has three elements: a condition, a set of state changes, and a set of outputs. We generate it based on this grammar:

```
<behavior> ⟶
  if <condition> then
    <state_changes>
    <outputs>
  end if;
```

```
<state_changes> ⟶ ϵ | <state_change> <state_changes>
```

```
<outputs> ⟶ ϵ | <output> <outputs>
```

State changes are modeled as assignments of the corresponding state variables:

```
<state_change> ⟶ <state_variable_name> := <new_value>;
```

```
<state_variable_name> ⟶ str
<new_value> ⟶ str
```

Message outputs are modeled by appending a new message to the queue of the corresponding output port:

```
<output> ⟶ <out_port_name> := <out_port_name> ⊕ msg
<out_port_name> ⟶ str
```

6.1.5 Topics

Topics are modeled as processes that take elements from the output queues and add them to all input queues of subscribers:

```
<topic> ⟶
fair process <topic_name> ∈ <topic_name.capitalize()>
  begin
    Write:
      if <topic_name> ≠ <> then
        msg := Head(<topic_name>);
        <topic_name> := Tail(<topic_name>);
        <in_ports_spec>
      end if;
  end process;
<in_ports_spec> ⟶ ϵ | <in_port_spec> <in_ports_spec>
<in_port_spec> ⟶ ϵ | <in_port_name> := <in_port_name> ⊕ msg;
<topic_name> ⟶ str
<topic_name.capitalize()> ⟶ str
<in_port_name> ⟶ str
```

6.2 Model Checking

After generating a TLA+ model from PlusCal using TLA+'s built-in functionality, the next goal is to find bugs in the resulting model. There are three main types of properties that can be checked on the generated TLA+ models:

Deadlock Freedom: To avoid a situation in which components get into a deadlock because they indefinitely wait for each other's messages, model-based analyses can check for deadlocks in the system. An analysis for system-wide deadlocks is built into TLA+ and can be activated with a simple check box.

Liveness Properties: To ensure that a certain type of desired system behavior happens, such as "eventually the planning component reaches the ready state", liveness property verification is needed. Since component states are specified directly in the TLA+ model, liveness properties can be specified

directly in TLC, e.g., "`<>A_ready = true`" to specify that component A should reach the ready state eventually. Specifying that component `B` should eventually send a message to topic `t` can be specified as the message queue of the output port being non-empty, e.g., "`<>(B_c /= «»)`". To specify that behavior can always happen (again) after any given point in the execution of the system, the operator for always eventually "`[]<>`" can be used. Since TLA+ allows us to verify arbitrary LTL properties via TLC and there will be a large variance of liveness properties to specify, we allow users to directly specify properties in TLC and use built-in verifiers to check them.

Causal Reaction Properties: To prevent bugs such as missing or inconsistent connectors, lost messages, or ignored inputs it is desirable to check properties that verify a causal relationship between an action and its reaction, such as an input A results in eventually sending a response B. The corresponding LTL property for this type of behavior is `[](A => <>B)`. TLC offers easier syntax for this: `A ~>B`. Using this type of syntax properties such as message A results in message B can be specified as: `a_out /= «» ~> c_in /= «»`.

6.2.1 Property Generation

To simplify checking the generated model for users who might not be formal modeling experts, we generate TLA+ properties automatically from a list of topics that are expected to receive at least one message. Users specify the list of topics, and then the corresponding liveness properties are generated automatically and added to the TLA+ model.

In future work, large language models could be used to generate more comprehensive TLC properties based on natural language specifications.

6.3 Real-World Bug Finding

To evaluate `ROSFindBugs`, to illustrate the variety of real-world architecture misconfiguration bugs, and to foster more research and evaluation on them, we constructed and provided a data set of bugs from real-world open-source ROS systems.
This data set can be used to study properties of architecture misconfiguration bugs and to evaluate bug finding techniques for these bugs.

6.3.1 A Data Set of Analyzable ROS Systems

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

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

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

We used the data set from Malavolta et al. [115] as a basis for the system selection. The resulting systems are: AutoRally [67], Autoware [94], Fetch [167], Husky, and TurtleBot [151].

6.3.2 A Data Set of Architecture Misconfiguration Bugs in ROS

Second, we identified bugs within the systems described in Section 6.3.1 with the following methodology:

Selection Criteria: We collected documented bugs on GitHub from repositories discussed in Malavolta et al. [115] (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 6.1, the data set contains 29 bugs across 5 systems. Note that this is not intended to be a complete list of architecture misconfiguration bugs in those systems.

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

Building Historic Project Versions: Some of the commits related to the bugs are many years old and were built with much older versions of their dependencies and much older versions of ROS (the oldest bug dates back to March 2014 and was reproduced with ROS Indigo Igloo). To support replicability, we created Docker images for each commit, each containing the versions of their build- and run-dependencies (including ROS packages, Compute Unified Device Architecture (CUDA) API version, external libraries, compilers, and the ROS distribution). If the project documented which versions of 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).

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

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

Table 6.2: Overview of the architectural misconfiguration bugs and whether our previous work `ROSDiscover` has detected the bug ("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.

To identify which of these bugs can be found with structural architectural recovery, we evaluated `ROSDiscover` on this data set with the following results shown in Table 6.2.

6.3.3 `ROSFindBugs`' Effectiveness

Three of the bugs described above (autoware-02, autoware-03, autoware-10) can be classified as *behavioral* architecture composition bugs. Autoware-02 is shown in Figure 3.2 (a). The other two also result from required inputs for important component behavior not being connected to publishers. We ran `ROSInfer` on these systems to infer models, generated PlusCal/TLA+ specifications via `ROSFindBugs`, and checked that expected outputs happen eventually. The resulting TLA+ models are shown in Listing A.1, Listing A.2, and Listing A.3, respectively. `ROSFindBugs` found all of these bugs based on a given list of components in the system configuration, a desired output to check for, and configuration parameter assignments. This shows that `ROSFindBugs` is practically useful at finding real-world behavioral architectural misconfiguration bugs that previous work could not find.

6.4 A Data Set of Behavioral Architecture Misconfiguration Bugs in ROS

Having identified that `ROSFindBugs` can find real-world bugs, the next piece of evidence for its usefulness is showing that these bugs are commonly found in real-world ROS systems and not just limited to the three bugs described above.

Therefore, in collaboration with Siyan Wu, we extended the data set to specifically include behavioral architecture composition bugs. Initially, we started a keyword search with the following keywords: "indefinite waiting", "infinite waiting", "fix mandatory input", "race condition", "state bug", "wrong state", "wrong frequency", "unexpected rate", "fix state", "fix rate", "message loss" "deadlock", and "fix queue size". This keyword search in the systems `husky`, `turtlebot`, `autoware`, `autorally`, `mavros`, and `jackal_robot`, containing 8572 commits in total, resulted in 861 keyword hits. We then manually verified that the bugs fit the definition, resulting in 18 verified behavioral architecture composition bugs. Out of those 18 bugs, we were able to build Docker images for 14 bugs.

To find more bugs beyond the literal matching of keywords in commit messages, we then used large language models (LLMs) to find bugs in the remaining commits. We provided the LLM with the commit message as well as the commit diff to use richer data for classification. We designed the prompt shown in Listing 6.1.

We ran the analysis with Llama 3 and added 6 more behavioral architecture composition bugs to the data set with this approach, bringing the total up to 20. The data set can be found here: https://docs.google.com/spreadsheets/d/1ymctYtsgl1CNWinbfUDQmGpbf9IbSXxvfMmi_mwPplA/edit?gid=0#gid=0. This shows that the class of bugs that `ROSFindBugs` can find commonly exists in real-world ROS systems.

6.5 Conclusions and Implications for the Dissertation

This chapter has shown how models inferred by `ROSInfer` can be automatically translated into PlusCal/TLA+ and then checked to find behavioral architecture composition bugs. This proves that assumptions about framework-specific APIs and idioms for ROS-based robotics systems enable the automatic inference of behavioral component models **that are useful for bug finding**. Further, we presented a newly collected data set of architecture misconfiguration bugs to show that these bugs commonly occur in ROS systems, providing further evidence that this work has practical value.

Listing 6.1: The prompt used to collect a dataset of behavioral architecture composition bugs.

You are a senior robotics software engineer specializing in advanced bug finding. You are checking whether a commit falls into the following description:
1. It describes or fixes a bug. This means it does not ONLY add new functionality and features, but deals with unexpected or incorrect behavior of the system.
2. The bug is a behavior architecture composition bug specifically related to inconsistent topic names, which manifests as incorrect or mismatched communication between components in a ROS-based system.
The key characteristics of an inconsistent topic name bug are as follows:

- The commit message contains certain keywords that indicate potential topic inconsistency, such as: [keyword-list]
- The commit affects files related to ROS topics, such as:
- If `.cpp` or `.yaml` files are changed together, this could indicate both a code change and a parameter change, potentially related to a topic renaming.
- Changes involve files related to `ros::Publisher` or `ros::Subscriber` code. Check if variable names related to topics have been altered.
- The changes involve only topic-related files or contain strings that match the format of ROS topics (e.g., `/topic_name`).

Note: The provided code changes follow the standard Git diff format. In the diff format:

- Lines starting with + indicate code that has been **added**.
- Lines starting with - indicate code that has been **removed**.

Your answer should be in the following format:
{
"answer": True/False,
"reason": {{ Short description of the reason for your judgement. }}
}

7 ROSView: Automatically Generating Behavioral Architectural Diagrams

Previous chapters described how to infer behavioral component models and how they can be used for bug findings. This chapter demonstrates how inferred models are useful for human developers in helping them understand the complex behavior and interactions of ROS components. We present `ROSView`, an approach to automatically visualizing components and their internal behavior in visual diagrams. Further, we evaluate the effectiveness of the diagrams in a study with real roboticists and robotics graduate students.

7.1 Motivation

Preventing and debugging behavioral architecture composition bugs requires a deep understanding of the behavior of inter-component communication and the state machines of interacting components. Understanding the behavior of ROS components can be challenging, as the code that defines architecturally-relevant component behavior is scattered throughout large code bases. While architectural diagrams (such as common component-connector models) can help visualize the connection of software components, most commonly used diagram styles are not optimized for visualizing state-based behavioral interactions. On the one hand, high-level perspectives that visualize components and their connections lack the granularity to support understanding behavioral assumptions. On the other hand, detailed perspectives that visualize a predominant portion of the behavior of a component contain a lot of information that might make it harder to focus on component interactions. Therefore, we propose a perspective of medium-grained detail that mixes elements of high-level structures with the behavior that is relevant for understanding how components interact and what assumptions they make about each other.

This chapter presents an approach that automatically generates visual diagrams that mix structural views of connected components with their architecturally-relevant behavior to provide information that helps developers understand the complex behavior of systems written for ROS.
This chapter describes:

1. `ROSView`: An approach to visualize component state machines inferred by `ROSInfer` (Section 7.2).
2. The design of the human study to evaluate the
3. The results of a human study with 24 roboticists.

7.2 Architectural Behavioral Diagrams

The approach we are presenting generalizes to component-port-connector architectures of systems whose behaviors can be described with reactive, periodic, and state-based behavior as formalized in Section 3.1.

To help developers understand architecturally-relevant component behavior, we enhance component-port-connector diagrams with behavioral elements inside each component. Component-port-connector

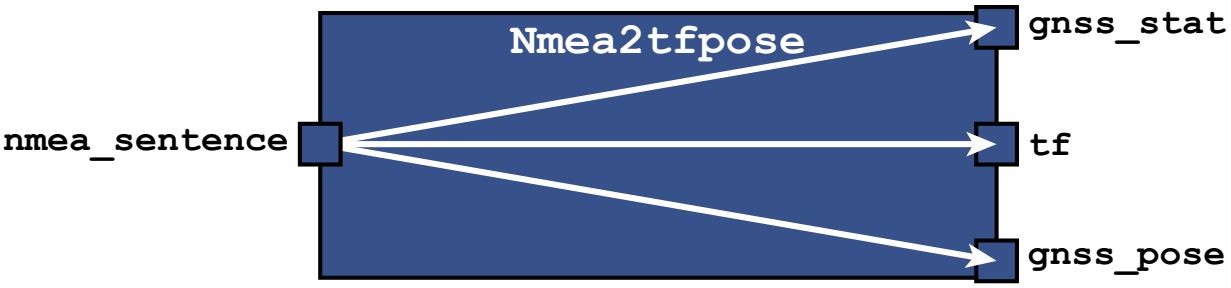


Figure 7.1: Nmea2tfpose Component

diagrams are a common visualization of run-time components (in ROS: nodes), their input ports (in ROS: subscribers), their output ports (in ROS: publishers), and connectors (in ROS: topics between subscribers and publishers) [40]. To enhance the understandability of the diagram, we position input ports on the left-hand side and output ports on the right-hand side of the component, following the common left-to-right reading direction.

We visualize periodic triggers as circular arrows with a frequency label in the center. State changes are visualized as via a label consisting of the name of the variable, and the assignment operator (`:=`) and the new value expression. We visualize initial states as assignment expressions in the top left of the component. Reactive behavior is visualized with an arrow starting from the trigger (input port or periodic trigger) and ending at the reaction (output port or state change). If reactive behavior is conditional, the condition is visualized via a label in square brackets in the center of the arrow.

7.3 Study Design

In this study, we are testing the following hypotheses:

H1 (Correctness) **Results in Section 7.4.1**

Hypothesis H1: Visual diagrams generated by `ROSView` increase the correctness of the understanding of the behavior of ROS components compared to only reading the source code.

This hypothesis is motivated by the idea that visual diagrams might help participants navigate the complexity of component behavior more easily and therefore might help them collect the information needed to answer the questions.

H2 (Task Completion Time) **Results in Section 7.4.2**

Hypothesis H2: Visual diagrams generated by `ROSView` reduce the time it takes to understand the behavior of ROS components compared to only reading the source code.

This hypothesis is motivated by the idea that participants would only need to check the parts of the source code that are needed to answer the portions of the questions that participants believe are not answered by the diagram, and therefore might reduce the time to read many lines of code.

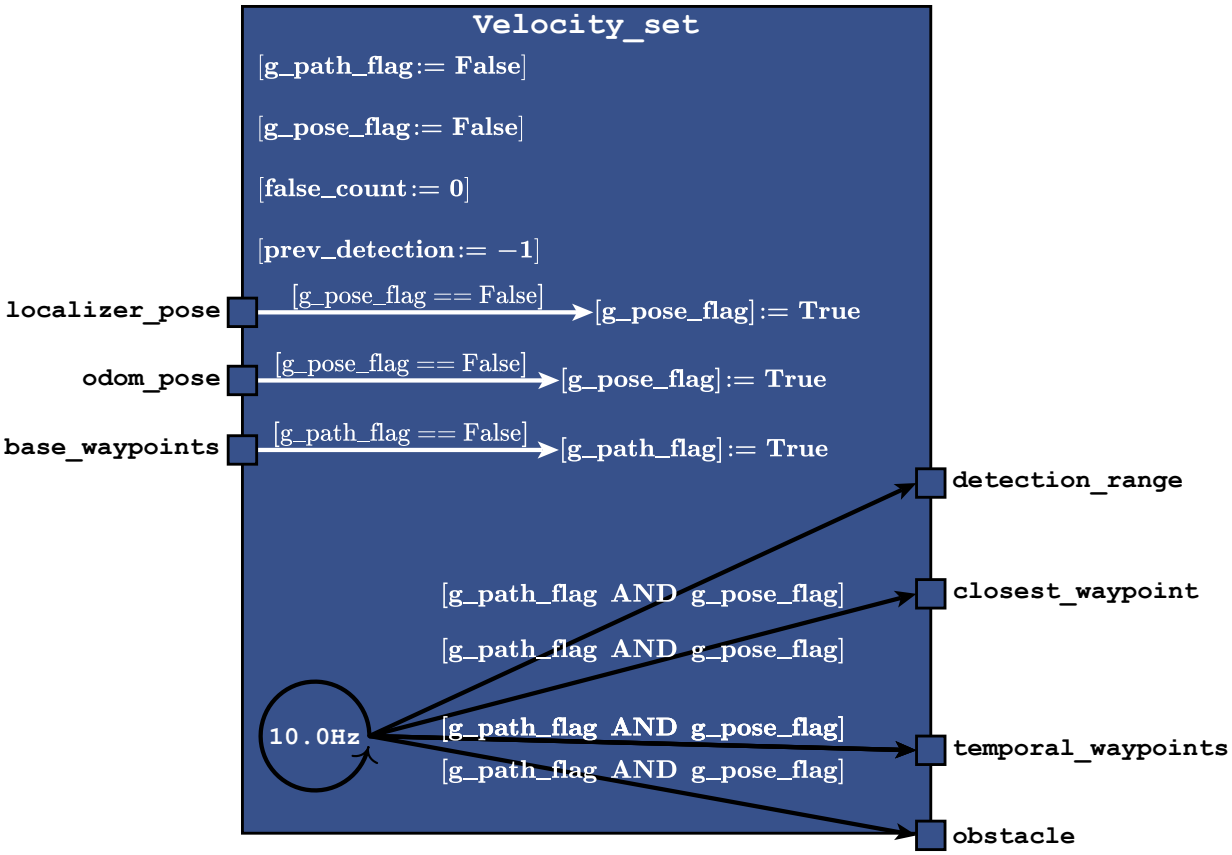


Figure 7.2: Velocity_set

H3 (Confidence) **Results in Section 7.4.3**

Hypothesis H3: Visual diagrams generated by `ROSView` increase confidence in the understanding of the behavior of ROS component compared to only reading the source code.

This hypothesis is motivated by the idea that participants might subjectively realize that their understanding of component behavior might be enhanced by visual diagrams.

Overall Design: To test these hypotheses, we designed a controlled experiment that lets participants answer questions on the behavior or ROS components with or without diagrams.

We randomly split participants into two groups. Both groups received the same three tasks. For each task, one group was presented with a diagram and the corresponding source code, while the other group only received the source code. To minimize the effect of individual preferences for code or diagrams, each group saw the diagram for at least one task (i.e., group 1 saw diagrams for task 1 and task 3; group 2 saw diagrams for task 2, with tasks alternating between code and diagrams to minimize potential training effects).

To ensure all participants are familiar with the notation of the diagrams, the tasks for both groups start with a tutorial explaining the notation and checking participants' understanding via verbally answered questions.

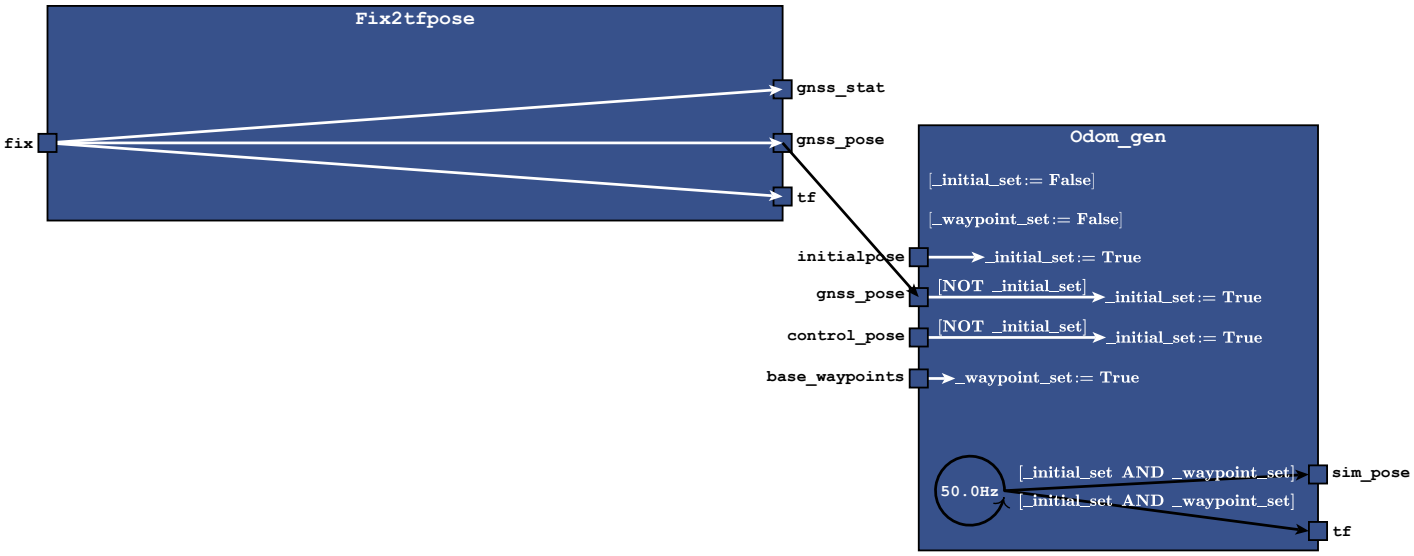


Figure 7.3: composition

Participants & Recruitment: We recruited participants with a background in C++ and experience using the ROS framework by advertising the study on X (formerly known as Twitter), LinkedIn, ROS Discourse, Mastodon, and via posters and emails within our university. To filter out potential participants without enough experience with ROS in C++, we asked three questions (which ROS API call do you commonly use to: (1) send a message, (2) define a publisher, (3) define the callback for receiving a message). We compensated participants with a 20 USD Amazon gift card.

In total, we had 24 participants (12 per group).

7.3.1 Tasks

We designed the tasks of the study to check participants' understanding of ROS component behavior. The study contained three main tasks with increasing complexity.

Tutorial: To ensure all participants understand the diagram notations, the study began with a tutorial. Participants saw a description of the semantics of the diagram elements based on an example component (Figure 7.4). They were also introduced to the concept of traces (*"A trace is the sequence of messages sent or received from the very beginning of the system execution. A trace is considered 'valid' if it describes one possible sequence of the messages that can be observed during the execution of the system."*) To check participants' understanding of the notation, we asked the following questions to be answered verbally: (1) What is the initial state of ready? (2) What causes `cmd_vel` to be sent? (3) Under what conditions are `cmd_vel` messages sent? (4) What is the consequence of receiving an `initial_pose` message? (5) Which of these traces are valid? Followed by three traces). If participants answered incorrectly, we re-explained the concept until they were able to answer correctly.

Task 1 - Output Behavior: We designed the first task to test whether participants can reliably identify output topics in the presence of wrapper API calls. We selected the Nmea2tfpose component (Figure 7.1), as it uses the very common ROS API call `tf::TransformBroadcaster::sendTransform` to send a message to the `tf` topic. The Nmea2tfpose component consists of only 186 lines of code, making it a

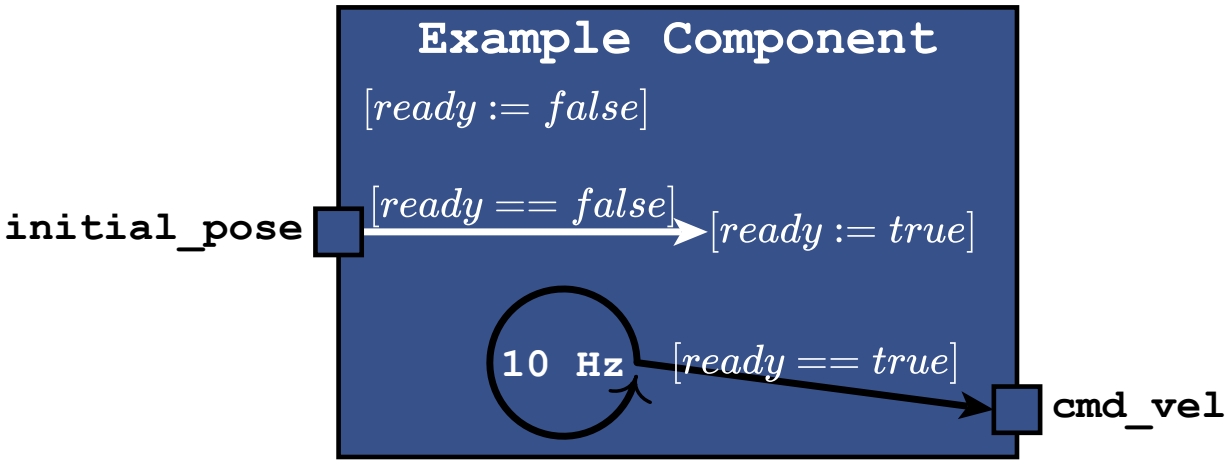


Figure 7.4: Example Component

simple component. We asked participants *"Which topic(s) does the component send messages to? Please list all applicable topic names"*, expecting three answers (`tf`, `gnss_pose`, and `gnss_stat`). Then we asked participants, *"What causes the component to send a message to the gnss_pose topic? Please answer this question concerning externally visible behavior of the component, not the corresponding method names."*, expecting two parts: (1) reactive to receiving an `nmea_sentence` message, (2) conditionally based on the content of the message. We selected this question, as it cannot be correctly answered *only* with the diagram, to identify how participants perform on tasks that require combining information from the diagram and code.

Task 2 - Complex Component Behavior: We designed the second task to test participants' understanding of complex, state-based component behavior. We selected the Velocity_set component (Figure 7.2), as it has periodic behavior that requires the component to receive two messages first to change to the message-sending state.

To correctly answer these questions, participants need to identify the following facts: (1) all output topics are published periodically only when the `gg_pose_flag_flag` and `g_pose_flag` are changed to true; (2) `gg_pose_flag` is changed to true when a `base_waypoints` is received; (3) `g_pose_flag` is changed to true when either a `localizer_pose` or `odom_pose` message is received; (4) obstacle messages are only sent when the component detects an obstacle in its input messages, which is not shown in the diagram; (5) obstacle detection only happens after `vs_scan` messages are received, which is not shown in the diagram;

The Velocity_set component consists of 958 lines of code, making it a fairly complex component.

Task 3 - Behavioral Architecture Composition Bug: We designed the third task to test whether participants can find and fix bugs more effectively with diagrams. We selected two interacting components (Fix2tfpose and Odom_gen (Figure 7.3)) and described observations of buggy behavior: *"The buggy behavior is that the Odom_gen component does not send any outputs, as seen in the system traces (e.g., bag files). Your colleague checked that the parameters are set correctly for Odom_gen component and observed that there are many instances of fix messages being sent. Considering this information, what could be a potential reason for Odom_gen not sending any messages?"* This description is designed to leave exactly one plausible explanation from an architectural behavior perspective: No component is sending messages to the

`base_waypoints` topic, as it is the only required topic for which there are no messages directly observed or caused by observed messages.

To answer this question correctly, participants need to identify the following facts: (1) both outputs of `Odom_gen` require the `_initial_set` and `waypoint_set` state variables to be true, (2) `_initial_set` is changed to true when Odom_gen receives a `gnss_pose` message, (3) `gnss_pose` messages are sent by the `Fix2tfpose` component when it receives `fix` messages, (5) `waypoint_set` is changed to true when `Odom_gen` receives a `base_waypoints` message, (6) `base_waypoints` messages are not sent by `Fix2tfpose`.

The `Odom_gen` component consists of 250 lines of code, while `Fix2tfpose` consists of 121 lines of code.

7.3.2 Threats to Validity

The results of this study have to be considered within the context of the following threats to validity.
Internal Validity: The order in which participants perform tasks might impact their responses. Participants' completion of previous tasks with diagrams and/or the initial tutorial might prime participants to think about tasks using the mental model of a component state machine and therefore lead to participants consciously or subconsciously trying to build the same representation that would be shown in the diagram of the alternative group. This threat to validity would decrease the observed effect of visual diagrams while at the same time increasing the task competition time for a subset of participants.

Furthermore, participants' self-assessment of their confidence, the helpfulness of diagrams, and their own expertise could be biased.

External Validity: With respect to generalizability to different ROS developers, the participants of the study might differ from the overall population of ROS developers, as most of them were in an early career stage. We attempted to reduce this bias by offering compensation to participants. However, experienced developers can be too busy to be willing to participate in research studies.

Furthermore, with respect to generalizability to other development tasks, the tasks of this study had to be minimize expected completion time. To minimize this threat, we varied the complexity between tasks. However, results might be different for tasks of higher complexity that would take multiple hours to complete.

7.4 Results

This section interprets the results for each of the three hypotheses.

The analysis of answers of the post-completion survey and notes from observing participants' thought process offer the following insights:

Most participants had intermediate ROS skills: 14 participants self-rated their experience with ROS as "Intermediate", while 5 rated them as "Advanced" and 5 as "Beginner". There was no statistical difference between the two groups (p=1.000).

Most participants had intermediate C++ skills: 17 participants self-rated their C++ skills as "Intermediate", while 4 rated them as "Advanced" and 3 as "Beginner". There was no statistical difference between the two groups (p=0.279).

Most participants had intermediate state machine skills: 15 participants self-rated their experience with state machines as "Intermediate", while 2 rated them as "Advanced" and 5 as "Beginner", and 2 as "Unfamiliar". There was no statistical difference between the two groups (p=0.940).

All participants perceived the diagrams to be helpful: To the question "How helpful or unhelpful was the provided diagram for answering the question(s)?" half of the participants answered "Having the diagram made it slightly easier to answer the questions" while the other half answered "Having the diagram made it much easier to answer the questions". There was no difference between the different groups. Nobody answered that "Having the diagram made no difference", or made it "slightly harder", or "much harder" to answer the questions. These subjective perceptions are further evidence that the objective performance differences between the two groups were, in fact, caused by the presented visual diagrams. Some participants confirmed this in their answer to the question "How did the diagram impact your decision?" One participant wrote "Yes, the diagram help me understand the structure and the behavior of the program better". Others wrote "I think they helped in getting [an] overview of what the behavior is supposed to be.", "[The diagram] helps me to understand the logic of the code", and "the diagram helped me have an understanding of the system and where I should look to isolate issues".

Some participants first looked at the diagram, while others started looking at the code: Participants varied in how they used the diagrams. Some participants used the diagram to get an overview, and they either identified which information was missing and searched for this in the code or used the code to confirm the hypothesis that they gained from the diagram. Representative quotes for this approach include "The diagram helped me get an overall system overview before i jumped into the code."

Other participants started with looking at the code and then looked at the diagram to summarize their insights, e.g., "[The diagrams] helped me confirm my thinking, which I originally got from the code".

This implies that participants used different strategies and/or saw diagrams as most useful for different aspects of their thought process.

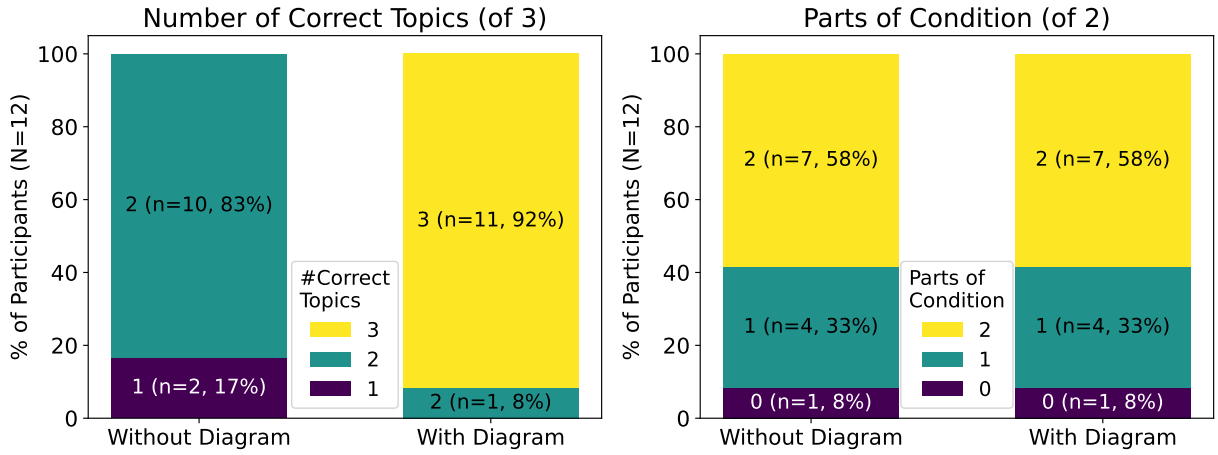
7.4.1 H1 - Correctness

To compare distributions statistically, we used the one-sided Mann-Whitney U test, which is a non-parametric test with the null hypothesis that one population's performance is not greater than the other populations performance.

Task 1 - Output Behavior: Participants with diagrams performed better at identifying simple output behavior. As shown in Figure 7.5 (a), participants with diagrams were able to list more correct output topics ($p<0.0001^{***}$, on average 1.08 more out of 3). The distribution of correct parts of the condition is identical for both groups (Figure 7.5 (b)). Since participants with diagrams did not take longer to complete the task (Figure 7.8 (a)), these results suggest that the shown diagram helped participants correctly and efficiently complete the task.

There is no difference in the distribution of participants' confidence rating between both groups (Figure 7.9 (a)).

Task 2 - Complex Component Behavior: Participants with diagrams performed better at understanding complex component behavior. As shown in Figure 7.6 (a), participants with diagrams were able to list more correct traces ($p=0.017^{*}$, on average 0.58 more out of 2). Furthermore, participants with diagrams identified more parts of the conditions ($p<0.0001^{***}$, on average 2.08 more out of 6) (Figure 7.6 (b)). Since participants with diagrams did not take longer to complete the task (Figure 7.8 (b)), these results suggest that the shown diagram helped participants correctly and efficiently complete the task.



(a) Correctly identified output topics (out of 3).

(b) Correctly identified parts of the condition (out of 2).

Figure 7.5: Distributions of response correctness for Task 1.

On average, participants with diagrams had higher confidence in their answers (Figure 7.9 (b)). Out of all three tasks, this task has the highest difference in confidence. One possible interpretation of this is that, particularly for understanding complex behavior, component state machine diagrams give participants more confidence in their answers as the diagrams give participants the sense of seeing relevant information.

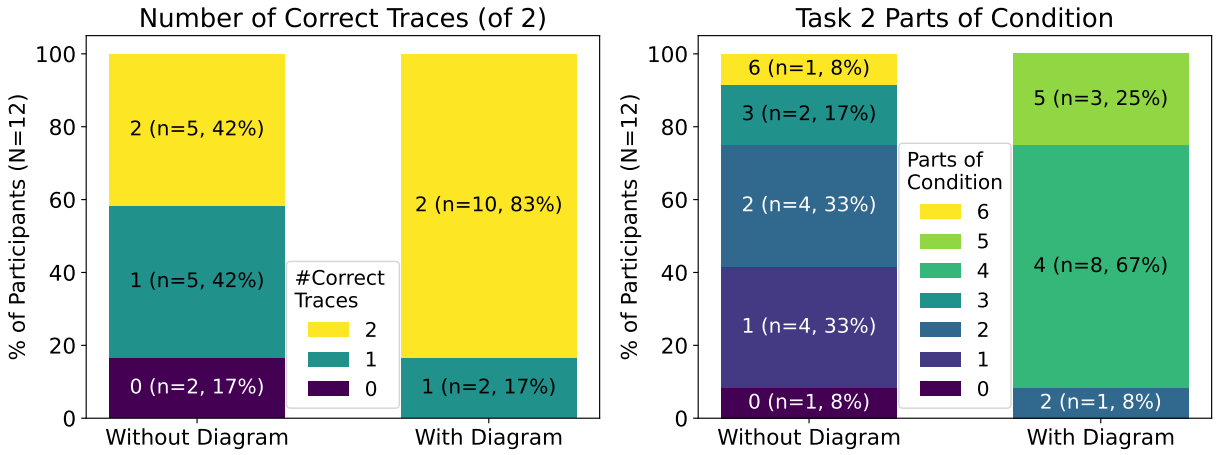
Task 3 - Behavioral Architecture Composition Bug: Participants with diagrams performed better at bug finding and bug fixing tasks while taking less time. As shown in Figure 7.7 (a), participants with diagrams were able to find more bugs (p=0.0032**). Furthermore, participants with diagrams were able to fix more bugs (p=0.0037**) (Figure 7.7 (b)).

Confirmation of H1 (Correctness)

In all three tasks, the correctness of answers by participants with diagrams was higher or equal to the correctness provided by participants without diagrams. Therefore, we can confirm hypothesis H1 (visual diagrams generated by `ROSView` increase the correctness of the understanding of the behavior of ROS components compared to only reading the source code.)

7.4.2 H2 - Task Completion Time

The measured results for task completion times are shown in Figure 7.8. For all three tasks, the mean task completion time for participants with diagrams was lower than for participants without diagrams. However, this difference was not statistically significant p=1.000 for task 1, p=0.015* for task 2, and p=0.591 for task 3). In the post-completion survey, two participants explicitly stated that they believe diagrams helped them answer the questions more quickly (e.g., "[The diagrams] saved me a lot of time to look through the entire code to figure out the trace"), suggesting that, subjectively these two participants thought that the diagrams reduced their completion time. However, this cannot be confirmed based on the data we collected in this study.



(a) Correctly identified traces (out of 2).

(b) Correctly identified parts of the condition (out of 6).

Figure 7.6: Distributions of response correctness for Task 2.

Rejection of H2 (Task Completion Time)

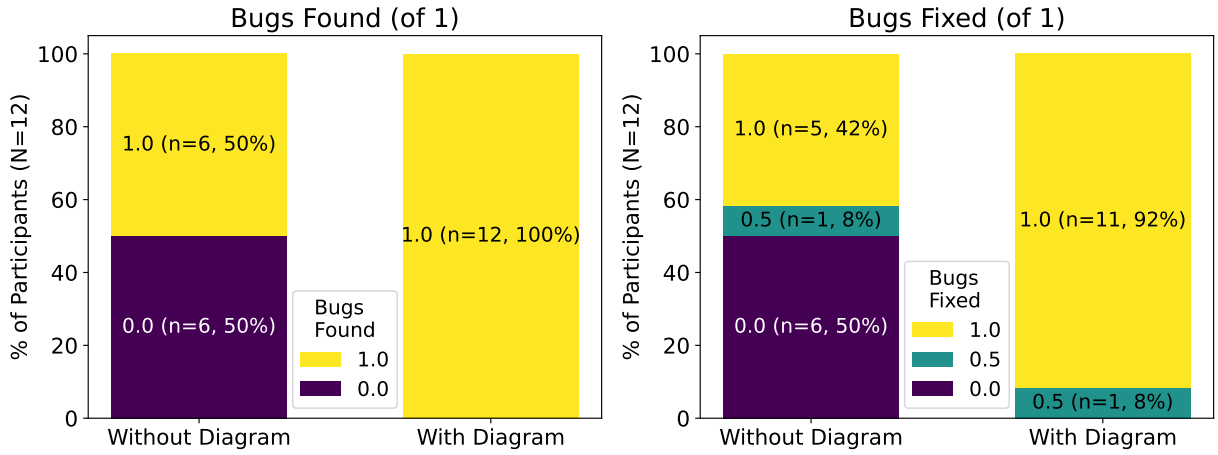
Due to the lack of statistical significance, we have to reject hypothesis H2 (visual diagrams generated by `ROSView` reduce the time it takes to understand the behavior of ROS components compared to only reading the source code). However, we can conclude that diagrams did not slow down participants in their tasks.

7.4.3 H3 - Confidence

The results of self-reported confidence is shown in Figure 7.9. For task 1, there was no difference in confidence. For task 2, participants with the diagram had higher confidence (p=0.015*). For task 3, participants with the diagram had slightly higher confidence, while the difference is not statistically significant (p=0.015*). In their post-completion survey, three participants noted that diagrams gave them more confidence (e.g., "The diagrams made me more confident after making conjectures based on looking at source code.")

Partial Confirmation of H3 (Confidence)

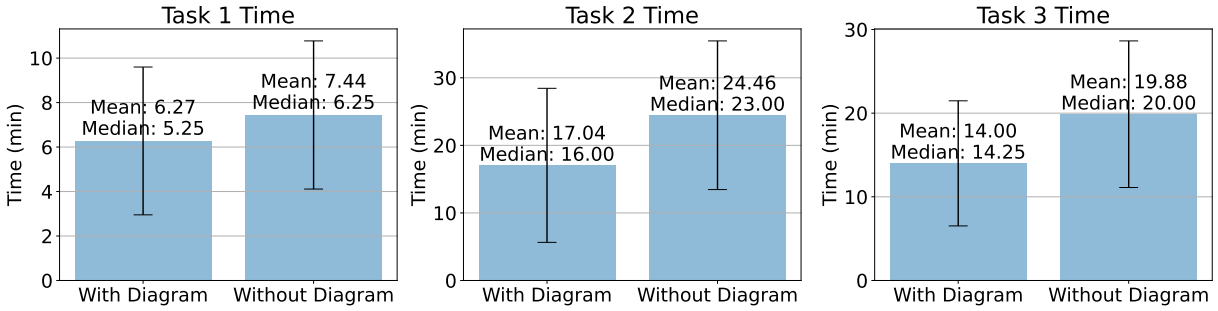
Since we observed a statistically significant improvement in confidence for one of the three tasks and no difference for the other two tasks while also receiving subjective feedback from some participants that diagrams made them more confident, we can partially confirm H3: visual diagrams generated by `ROSView` **sometimes** increase confidence in the understanding of the behavior of ROS components compared to only reading the source code.



(a) Participants' performance for the bug finding question (1 means bug found, 0 means not found)

(b) Participants' performance for the bug fixing question (1 means bug fixed , 0 means not fixed, 0.5 means half fixed)

Figure 7.7: Distributions of response correctness for Task 3.



(a) Task 1: p=0.403

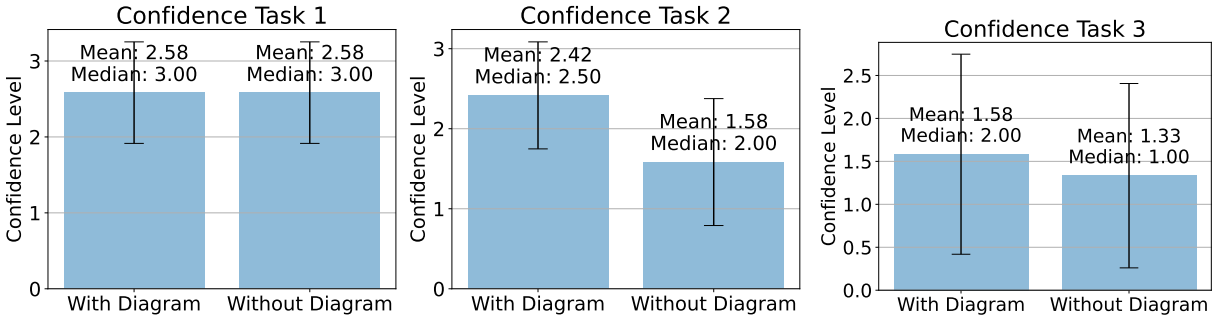
(b) Task 2: p=0.184

(c) Task 3: p=0.119

Figure 7.8: Distribution of task completion times per task in minutes.

7.5 Conclusions and Implications for the Dissertation

This chapter has shown that the models inferred by `ROSInfer` can be automatically translated into visual diagrams. The results of the human study have shown that the diagrams help roboticists understand the behavior of complex components and their interactions, as they result in more correct answers while at the same time now slowing down participants and not reducing the confidence in their answers. This provides evidence of the practical usefulness of the inferred models, which proves the last piece of thesis statement.



(a) Task 1: p=1.000 **(b)** Task 2: p=0.015* **(c)** Task 3: p=0.591

Figure 7.9: Distributions of self-reported confidence in provided answers per task.

8 Discussion & Conclusions

This chapter compares the presented approaches across multiple dimensions, identifies limitations, discusses how useful the particular approaches are within different contexts, and argues that the work presented in this dissertation is a significant contribution to the field of software engineering.

8.1 Static Analysis & Dynamic Analysis

The presented static analysis (Chapter 4) and dynamic analysis (Chapter 5) are both approaches to infer behavioral models from existing systems. However, they come with different trade-offs.

Assumptions & Inputs: Static analysis requires fewer inputs and makes fewer assumptions about an available execution environment. The inputs for static analysis are the source code and launch file configurations. Dynamic analysis additionally requires a representative execution environment (e.g., the real hardware or a simulation environment) and a sequence of inputs to run the system. This implies that dynamic analysis needs more effort to create the initial setup. It also requires more maintenance effort when the system is evolving (input sequences might need to be updated, and the execution environment might change). Furthermore, for complex components, the execution time of dynamic analysis will generally take longer.

Soundness & Limitations: Static and dynamic analysis have different strengths and weaknesses by design, but also depend on how the code is structured.

Since static analysis (at least in theory) has access to the complete program behavior, while dynamic analysis can only observe a finite set of program executions, static analysis has the potential to provide higher confidence in the output. In particular, the following aspects of `ROSInfer`'s analysis are sound (but not complete, as the results can sometimes be $\top$):

Periodic Target Frequencies: Since the frequencies inferred by `ROSInfer` are direct arguments to ROS API calls, if constant folding succeeds, the inferred frequency is a sound upper bound of the frequency of the behavior (it is not a lower bound, as the frequency can be lower in overload situations). In contrast, dynamic analysis can only finitely sample instances of the frequency value. If the value is context-dependent (e.g., it is determined based on values in messages or other external data sources), then dynamic analysis would miss cases in which the frequency could have smaller or larger values. While static analysis would soundly identify these cases as known unknowns, the results of dynamic analysis would include unknown unknowns and, therefore, be unsound.

Message Triggers: If `ROSInfer` finds a publish call and identifies that it is (transitively) called only within a subscriber callback, `ROSInfer` can conclude that receiving this message has to be part of the pre-condition of the corresponding publishing behavior. Note that `ROSInfer` might miss some other conditions that have to be true and hence does not guarantee completeness. In other words, it can partially determine causality in the sense that the message trigger is a necessary, but not sufficient, part of the publishing behavior's condition. The same applies to state changes

being triggered by messages via the identification of state variable assignments (transitively called) within a subscriber callback.

The following aspects of `ROSInfer`'s analysis are not guaranteed to be sound:

Periodic Behavior: The inference of periodic behavior is not guaranteed to be sound, as `ROSInfer`, for example, misclassifies looping over sequence data structures as periodic, even though it sends a fixed number of messages.

Reactive Behavior: The inference of reactive behavior is not guaranteed to be sound, as `ROSInfer`, for example, misclassifies behavior that reacts to a trigger that is outside of its scope (e.g., serial inputs) as reacting to the initial system startup.

State Variables: The inference of state variables is not guaranteed to be sound, as `ROSInfer`, for example, misclassifies parameter reads as state variables.

The main strength of these two main contributions lies in the combination of static and dynamic analysis, which can result in overall better models than each individually. Furthermore, we have shown that despite their limitations, the resulting models inferred by `ROSInfer` & `ROSInstrument` are still useful in practical settings, particularly for finding real-world bugs and for automatically generating visual diagrams that help developers understand complex software components. This implies that, even though the analysis is not guaranteed to be sound or complete, the resulting contribution is still significant, due to its measured effectiveness for real-world tasks.

To increase the effectiveness beyond the results shown in this dissertation, future work could handle more corner cases of static analysis and improve the coverage of dynamic analysis. For example, `ROSInfer` could be extended to model the internal state of arrays and dictionary data structures to support static analysis of more dynamic code structures. `ROSInstrument` could be extended to use Fuzzing techniques [116] to increase coverage. This additional engineering effort, in the order of months of work, could result in models with a higher recovery rate.

Use Cases & Workflow Integration: Neither static inference nor dynamic completion of models has soundness guarantees for all aspects of the models. Therefore, neither is well-suited for cases in which formal guarantees are required, such as safety analysis or certification. However, as they can still find bugs, they can reduce the cost of detecting bugs. The earlier bug detection can also reduce bug-fixing costs.

Based on the general observation that filling in values with dynamic analysis results in loss of soundness of some model properties, particularly in cases in which the system architecture is changing very dynamically and has many different configurations, it is less appropriate for use cases in which an architect wants to reason over, for example, a large product line. In these cases, manual model completion would be advisable over dynamic completion. On the other hand, dynamic analysis increases the overall amount of data in the models without much human intervention. So, in cases in which the system architecture does not vary greatly, it can result in more useful models.

8.2 Model Checking & Visual Diagrams

Model checking (Chapter 6) and visual diagrams (Chapter 7) both address the problem of finding bugs and demonstrate the usefulness of the inferred models. Their different capabilities and required effort

determine which use cases they are more useful for and how they integrate into a usual software development workflow.

Capabilities: Model-checking can be used to check a set of pre-defined properties that a senior engineer or architect identified to be expected for all system executions. Therefore, model-checking is well-suited for finding bugs that violate common architectural rules of expected behavior.

On the other hand, visual diagrams are particularly useful for checking whether the resulting architecture is what developers intended and for helping developers understand previously unknown components to prevent bugs in the first place.

Required Effort & Resources: Both approaches require the setup for static (and optionally dynamic) analysis as described above. In addition to this, model checking requires a specification of the desired behavior to be checked. Visual diagrams just require a specification of which components should be included in each desired view. However, to serve the purpose of bug finding, visual diagram inspection takes more developer time.

Use Cases & Workflow Integration: Since model-checking, once correctly configured, can run without human intervention, it can easily be integrated into the Continuous Integration (CI) pipeline of a project and can be executed after every pushed change in combination with existing test cases. Due to the additional human inspection time, in practice, bug finding with visual diagrams inspection might not be doable after every single change. While they might help developers identify if their changes had the architectural impact that they expected, the more immediate use case of diagram inspection would be before a public release, as a mechanism for debugging if the developer encounters unexpected component behavior, or to gain a better understanding of components that the developer is not familiar with.

8.3 Conclusions

In this dissertation, we have shown that despite their limitations, the resulting models inferred by `ROSInfer` & `ROSInstrument` are useful in practical settings. In particular, our results have shown that they can be used to find real-world bugs, and they can be used to generate visual diagrams that have helped developers understand the behavior of complex ROS components. As automated bug-finding and understanding complex interactions of components are challenging tasks that are particularly important in a society that increasingly integrates robots, this dissertation is a significant contribution towards ensuring the safety and software quality of robotics systems.

8.4 Design Education

Based on the insights from the visual diagrams study, that different developers have different approaches to understanding the code and diagrams, we hypothesize that dedicated software design education could also address the challenge of making complex component-based systems safer and more reliable. While automated bug-finding can have great economic benefits and can result in overall safer and more reliable robots, systematic design education has the potential to have an even larger impact, as it has the potential to prevent bugs in the first place. Furthermore, design education has the potential to increase the usefulness of the presented approaches, as it fosters a mindset that can more systematically engage with generated visual diagrams and formal models. However, recent graduates often lack important software design skills, such as generating, effectively communicating, and evaluating design options, and collaborating

across teams to build large systems [141, 63, 22]. We present an approach to educating students on how to design complex component interactions and our initial experience with this course at Carnegie Mellon in Appendix B as an additional minor contribution of this thesis. We propose the *GCE-paradigm* (i.e., the process of iteratively generating, communicating, and evaluating design options) as a guiding framework to systematically teach software design. Overall, the course has been well-received by the 17 students. They particularly valued the use of real-world case studies and in-class discussions. The multi-team project gave students insightful learning opportunities on cross-team communication that are rarely found in university education. Using interface descriptions and test double components, students could successfully integrate separately developed components. While most students' performance improved throughout the semester, some students continued to struggle with generating multiple viable alternatives and clearly communicating them via appropriate abstractions. Based on our lessons learned, we discuss recommendations to improve the course. The complete contribution can be found in Appendix B. Future work could empirically test whether this course design results in fewer architecture misconfiguration bugs in practice by running a controlled experiment with students who received this educational approach as an intervention and students who did not. Then measuring how many architecture misconfiguration bugs they produce in their future projects would indicate the effectiveness of this approach for reducing architecture misconfiguration bugs in the first place.

8.5 Future Work

The work in this dissertation motivates additional work that builds on `ROSInfer`, `ROSInstrument`, `ROSFindBugs`, `ROSView`. We envision the main contributions to enable the following future work:

ROS 2 Support: The presented implementations are specific to ROS 1. However, the approach could be extended to also support ROS 2. This would involve adding additional API calls from the ROS 2 API, parsing ROS 2 launch files and Python launch scripts, and looking for life cycle nodes to infer component state machines defined using the ROS 2 API. Adding ROS 2 support would replace the static analysis and dynamic analysis while keeping the component format identical. Thereby, the TLA+ generation, model-based analysis, and visual diagram generation could be reused without major changes. This work would likely be of the scope and size of a single research paper.

Generalization to Other Frameworks and Domains: The approach of API-call-based static & dynamic inference of component behavior is not inherently specific to the ROS framework but is likely to generalize to other domains as well. The main assumptions of API-call-guided recovery of component behavioral models are that (1) the domain is inherently component-based, i.e., most software is written in independently deployable run-time units that mostly communicate via messages; (3), the domain has a common framework that offers a well-defined API for components, component interactions, and component behaviors, (3) components, their interections, and behavioral patterns in real-world software in the domain are almost exclusively implemented using the framework API and common idioms. Other frameworks and domains that would satisfy these assumptions include, but are not limited to, NASA's FPrime framework [25] for small-scale flight software, the Kubernetes framework for microservices, and the Enterprise JavaBeans framework for component-based Java applications. To apply this approach to other domains, someone would need to identify the API calls that are equivalent to ROS API calls identified in this dissertation, implement a static analysis, a dynamic analysis, and potentially adjust the

semantics of the component-based model to the particular framework behavior. This work would likely be of the scope and size of a PhD thesis.

Automated Program Repair: Since models that were inferred from source code have the advantage of retaining a mapping between source code locations and model elements, a repair patch for the model could be translated back to code. This motivates future work on model-repair translations back to code. Counter-examples generated by TLA+ could be particularly useful in patch generation. In cases in which complex code transformation needs to occur, LLMs could be used to generate patches based on the provided counter-examples, models, and expected behavior. This work would likely be of the scope and size of a single research paper.

Repository Mining: Automatic inference of component behavioral models enables large-scale empirical research on the development and evolution of component behavior and inter-component communication patterns in complex robotics systems. To adjust `ROSInfer` to run on arbitrary GitHub code, it could be replaced with a simpler static analysis based on srcML [41]. Then, this analysis could be used to answer research questions, such as "How much can ROS architectures change during run time?". The assumption that ROS architecture does not change often has been made by many static analysis tools, such as `ROSDiscover` [158] and HAROS [144]. This assumption could be verified using a large-scale repository mining study by analyzing the percentage of architecture-defining ROS API calls that happen inside of conditions or loops.

Other research questions that could be answered using a large-scale repository mining study by measuring the distribution of metrics calculated on architectural models inferred using an approach similar to `ROSInfer` include: "What percentage of ROS components include periodic behavior?", "What percentage of ROS components include state-based behavior?", "How complex are state machines of ROS components?", etc.

This work would likely be of the scope and size of one or more research papers.

Transitive Pre-Condition Analysis: The questions "which messages have to be sent to component X so that it sends message X" and "which messages have to be sent to component X so that it changes to state Y" are often interesting to identifying what is needed to correctly configure the component. This analysis can be done on component behavioral models by identifying the pre-conditions of message sending behavior and the transitive pre-conditions of behaviors that would lead to the satisfaction of these pre-conditions. This work would likely be of the scope and size of a short paper.

Test Generation: Component behavioral models could be used to enhance test generation. Information about which input messages change the component to a state in which it executes different behavior can be helpful to systematically generate test cases to cover a larger portion of the component's behavior. This work would identify the sequences of messages that would result in covering all of the identified component states (based on the pre-condition analysis described above) in the model and create test cases that send these messages to the component. This work would likely be of the scope and size of a single research paper.

Automatic Generation of Documentation: Automated inference of behavior models can support the generation of documentation for components, especially for reusable components. In cases in which components need to receive a set of initialization inputs to function properly (such as the example from Figure 3.2 (a)), component documentation can be useful for developers who reuse existing components without being familiar with their internals. To further support developers beyond visual diagrams, future work could automatically generate documentation of, for example, required inputs, periodic outputs and their frequencies, and connected components. Required inputs can be inferred via pre-condition analysis

of the major behavior of the component. Information on periodic behavior and connected components is directly captured in component behavioral models. This work would likely be of the scope and size of a short paper.

Architectural Change Impact: When developers make major changes to the architecture of the system, an approach building on `ROSView` could visualize the impact of the change via a before-and-after visualization of impacted components and highlighted changes. To identify which components should be visualized, this technique would map the source files included in the code patch to the components, and visualize the components whose source code has changed, as well as their immediate connections. This visualization could help developers check if their changes had the intended effect and potentially prevent bugs earlier. This work would likely be of the scope and size of a single research paper.

Part III

Appendix

A Example Models

A.1 Autoware-02

```
------------------------------ MODULE autoware02 ------------------------------
EXTENDS Sequences, Integers, TLC, FiniteSets
CONSTANTS Lattice_trajectory_gen, Control_pose, Odom_pose, Base_waypoints, Data,
    NULL, MaxQueue

ASSUME NULL ∉ Data


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

(*--fair algorithm polling
variables
      cubic_splines_viz = <>;
  control_pose = <>;
  control_pose_lattice_trajectory_gen = <>;
  odom_pose = <>;
  odom_pose_lattice_trajectory_gen = <>;
  unknown_topic = <>;
  base_waypoints = <>;
  base_waypoints_lattice_trajectory_gen = <>;
  g_sim_mode = TRUE;

    define
        TypeInvariant ==
  cubic_splines_viz ∈ SeqOf(Data, MaxQueue) ∧
  control_pose ∈ SeqOf(Data, MaxQueue) ∧
  control_pose_lattice_trajectory_gen ∈ SeqOf(Data, MaxQueue) ∧
  odom_pose ∈ SeqOf(Data, MaxQueue) ∧
  odom_pose_lattice_trajectory_gen ∈ SeqOf(Data, MaxQueue) ∧
  unknown_topic ∈ SeqOf(Data, MaxQueue) ∧
  base_waypoints ∈ SeqOf(Data, MaxQueue) ∧
```

```
base_waypoints_lattice_trajectory_gen ∈ SeqOf(Data, MaxQueue)

    Response == <>(cubic_splines_viz ≠ <>)
  end define;

  fair process lattice_trajectory_gen ∈ Lattice_trajectory_gen
    variables
  msg ∈ Data;
  g_waypoint_set = FALSE;
  g_pose_set = FALSE;

    begin

            10_0Hz:
                if ((~(g_waypoint_set = FALSE)) ∧ (~((g_waypoint_set = FALSE)
  ∨ (g_pose_set = FALSE)))) then

                    unknown_topic := unknown_topic ⊕ msg;
                    cubic_splines_viz := cubic_splines_viz ⊕ msg;

                end if;

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

                        if TRUE then
                            g_waypoint_set := TRUE;


                        end if;

            end if;

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

                        if TRUE then
                            g_pose_set := TRUE;
```

```
                        end if;

            end if;

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

                        if g_sim_mode then
                            g_pose_set := TRUE;


                        end if;

            end if;

    end process;

fair process control_pose ∈ Control_pose

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

            end if;
    end process;

fair process odom_pose ∈ Odom_pose

    begin
        Write:
            if odom_pose ≠ <> then
                msg := Head(odom_pose);
                odom_pose := Tail(odom_pose);
                    odom_pose_lattice_trajectory_gen :=
odom_pose_lattice_trajectory_gen ⊕ msg;
```

```
                end if;
        end process;

    fair process base_waypoints ∈ Base_waypoints

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

                end if;
        end process;

end algorithm; *)
==
```

Listing A.1: Example TLA+ Code

A.2 Autoware-03

```
------------------------------ MODULE autoware03 ------------------------------
EXTENDS Sequences, Integers, TLC, FiniteSets
CONSTANTS Velocity_set, Localizer_pose, Odom_pose, Base_waypoints, Data, NULL,
    MaxQueue

ASSUME NULL ∉ Data


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

(*--fair algorithm polling
variables
      temporal_waypoints = <>;
  localizer_pose = <>;
  localizer_pose_velocity_set = <>;
```

```
obstacle = <>;
odom_pose = <>;
odom_pose_velocity_set = <>;
closest_waypoint = <>;
detection_range = <>;
base_waypoints = <>;
base_waypoints_velocity_set = <>;

  define
      TypeInvariant ==
temporal_waypoints ∈ SeqOf(Data, MaxQueue) ∧
localizer_pose ∈ SeqOf(Data, MaxQueue) ∧
localizer_pose_velocity_set ∈ SeqOf(Data, MaxQueue) ∧
obstacle ∈ SeqOf(Data, MaxQueue) ∧
odom_pose ∈ SeqOf(Data, MaxQueue) ∧
odom_pose_velocity_set ∈ SeqOf(Data, MaxQueue) ∧
closest_waypoint ∈ SeqOf(Data, MaxQueue) ∧
detection_range ∈ SeqOf(Data, MaxQueue) ∧
base_waypoints ∈ SeqOf(Data, MaxQueue) ∧
base_waypoints_velocity_set ∈ SeqOf(Data, MaxQueue)

      Response == <>(closest_waypoint ≠ <>)
  end define;

  fair process velocity_set ∈ Velocity_set
      variables
  msg ∈ Data;
  g_path_flag = FALSE;
  g_pose_flag = FALSE;
  false_count = 0;
  prev_detection = -1;

      begin

              10_0Hz:
                  if ((~(g_pose_flag = FALSE)) ∧ (~((g_pose_flag = FALSE) ∨ (
  g_path_flag = FALSE)))) then

                      detection_range := detection_range ⊕ msg;
                      temporal_waypoints := temporal_waypoints ⊕ msg;
                      closest_waypoint := closest_waypoint ⊕ msg;
                      obstacle := obstacle ⊕ msg;

                  end if;
```

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

                        if (g_pose_flag = FALSE) then
                            g_pose_flag := TRUE;


                        end if;

            end if;

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

                        if (g_pose_flag = FALSE) then
                            g_pose_flag := TRUE;


                        end if;

            end if;

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

                        if (g_path_flag = FALSE) then
                            g_path_flag := TRUE;


                        end if;

            end if;

    end process;
```

```
    fair process localizer_pose ∈ Localizer_pose

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

                end if;
        end process;

    fair process odom_pose ∈ Odom_pose

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

                end if;
        end process;

    fair process base_waypoints ∈ Base_waypoints

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

                end if;
        end process;

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

Listing A.2: Example TLA+ Code

A.3 Autoware-10

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

ASSUME NULL ∉ Data


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

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

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

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

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

begin

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

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


elsif ready_ then
isReady_obj_pos_xyz := TRUE;


end if;

end if;

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

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

```
elsif TRUE then
isReady_ndt_pose := TRUE;


end if;

end if;

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

if TRUE then
ready_ := TRUE;


end if;

end if;

end process;

fair process image_obj_tracked ∈ Image_obj_tracked

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

end if;
end process;

fair process current_pose ∈ Current_pose

begin
Write:
if current_pose ≠ <> then
msg := Head(current_pose);
current_pose := Tail(current_pose);
current_pose_obj_reproj := current_pose_obj_reproj ⊕ msg;
```

```
end if;
end process;

fair process unknown ∈ Unknown

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

end if;
end process;

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

Listing A.3: Example TLA+ Code

B Teaching Multi-Component Software Design Using Multi-Team Projects

The previous chapters presented approaches to automatically find bugs that result from the incorrect composition of software components and an approach help developers to find these bugs in visual diagrams. This chapter presents an approach to prevent these bugs by educating students to systematically design, implement, and test system that are composed out multiple interacting components.

B.1 Introduction

Designing software systems is an essential technical software engineering skill [14, 160, 6] that includes generation, communication, and evaluation of design options and working across teams to build complex multi-component systems [45, 137, 154, 86].

However, recent graduates often lack important software design skills, such as generating and comparing alternative designs, communicating them effectively, and collaborating across teams [141, 63, 22]. Multi-national, multi-institutional experiments have shown that the majority of graduating students in computer science lack the skills to design software systems [51, 111]. This gap between industry-needed competencies [14, 160, 6] and the design skills of recent graduates has also been confirmed by surveys of software practitioners [6].

This skill gap motivates a larger emphasis on software design education in universities [65, 63]. In many cases, software design is taught as just a small part of an overall software engineering course [6, 156, 133]. However, general software engineering courses give students little instruction and insufficient practice of software design skills in projects that are large enough to expose students to practical design challenges [134, 76, 141, 22]. In the cases in which software design is taught in a dedicated course, learning objectives focus on design patterns and architectural styles [134], which are important concepts for producing high-quality design artifacts. However, in contrast to design as an *artifact*, design as an *activity* [45] is rarely taught as a primary course objective [134, 17]. Therefore, students often lack the skills and mindset to systematically design complex software [141, 63, 22].

Teaching software design activities is challenging. Instructors have to find the right balance between teaching theoretical knowledge while also allowing students to gain enough practical experience with applying the taught design techniques to a realistic and sufficiently complex system [58, 76, 157, 134, 108]. In small software projects, students do not experience the challenges and learning opportunities that arise when no single person can fully understand the entire system [43, 42], such as compatibility of independently developed components [61], cross-team communication, component responsibility assignments, and workload distribution. Therefore, we believe software design is most effectively taught with a large-scale multi-team project that closely simulates the complexity and challenges of professional software development projects.

In this chapter, we present our experience designing and delivering a new course that teaches undergraduate and graduate students how to design large-scale software systems via case-study-driven lectures and a semester-long multi-team project. We propose the "*GCE-paradigm*" (i.e., the process of iteratively

***g**enerating*, ***c**ommunicating*, and ***e**valuating* design options) as a guiding framework to systematically teach software design activities. In lectures, students learn design principles based on positive and negative real-world case studies using constructivism learning theory [15] and active learning [26]. Further, we teach multi-team software design using interface descriptions and test double components. In the course project, student teams collaboratively design, implement, test, and integrate a large-scale multi-service web application and describe important design decisions in milestone reports.

Based on our lessons learned, we discuss recommendations to improve the course design. Overall, the course was well-received by students. 17 students across 4 teams successfully designed and implemented a complex system. Most students' performance in design activities improved throughout the semester. However, some students continued to struggle with generating multiple viable alternatives and clearly communicating them via appropriate abstractions. This suggests that students need more formative assessments and more concrete guidelines for these design activities.

B.2 Related Work on Software Design Education

B.2.1 Software Design Courses

Due to the importance of software design skills, courses on software design have been taught for decades [134, 150].

Lecture-focused Courses: Many software design courses in the literature focus on lecture-based learning without a major project component [134]. Some courses focus on teaching software design based on design patterns and remain closer to a source code [85, 165]. Other courses focus on high-level component interactions, architectural styles, and quality attributes [118, 62]. While these courses teach important skills that are relevant to producing good design artifacts, to the best of our knowledge, only one course at UC Irvine [17] teaches software design primarily as a systematic activity [45].

Team-Project-based Courses: Some software design courses include a major team-project component [134, 150]. For example, in a course taught at Murdoch University, students practice modular decomposition and learn to specify component interfaces in teams of six [13, 83]. UC Irvine includes two team projects in their software course during which students design and implement a system in teams of 14 students [17]. Courses taught at the University of Queensland [36] and Beihang University [172] provide students with open-source systems that students should read, model, and extend. A common domain for team projects in software design courses is game projects [169]. In existing software design courses student teams generally work individually, rather than collaboratively developing a system across teams. In contrast, our course allows students to experience cross-team communication challenges and a more realistic development context in which students have to integrate components built by other teams.

B.2.2 Multi-Team Courses

The teaching concept of using multiple interacting teams in software engineering education has been proposed and implemented in courses not focused on software design before.

Agile Processes: A course on scaling Scrum, which has been taught for multiple years at Hasso Plattner Institute, lets students build a web application with multiple interacting teams [124, 123]. The course teaches the Scrum process and modern software engineering practices (e.g., test-driven development, behavior-driven development, continuous integration, and version control) in a realistic environment with

self-organizing teams in a semester-long project [124]. Students receive the role of either Scrum Master, Product Owner, or developer while customers are simulated by the teaching team [123]. Students learn by making decisions about their development process autonomously and reflecting on their decisions after each sprint [124]. The multi-team project of our course has been partially inspired by this course. Similar courses are taught at the College of William and Mary [43, 42], the University of Helsinki [113], and the University of Victoria [107]. However, in contrast to multi-team courses on Agile processes, the learning objectives of our course focus on software design activities. This creates additional challenges, as the time available for teaching development processes and interactions is more limited in a software design course.

Global Software Development: Some courses teach even harder-to-practice skills of developing a product via collaborating, globally distributed teams [35, 39, 47, 75]. However, similar to the courses on Agile processes, they do not specifically focus their learning objectives on software design.

B.3 Course Design Overview

The course presented in this dissertation is a full-semester elective aimed at graduate and undergraduate students in computer science and majors related to computer science (e.g., information systems). Prerequisite knowledge of the course included intermediate programming skills and experience with developing and testing medium-sized programs. The course builds on the programming skills that students have obtained through previously taken courses, internships, or other industry experience and teaches them the highly demanded skills of designing large-scale software systems by making trade-offs between different quality attributes, considering different design alternatives, and communicating design using appropriate models. The course consists of three major instructional methods:

1. Active-learning-style **lectures** using real-world case studies to teach design principles based on constructivism learning theory [15] (Section B.4).

2. A semester-long **multi-team project** in which all teams collectively design, implement, and integrate a system composed of different services and describe their design decisions in five milestone reports (Section B.5).

3. Three **individual homework** assignments during which students practice skills taught in the lectures (Section B.6).

B.3.1 Learning Objectives (LOs)

As few existing courses teach software design primarily as an activity, deciding what to teach in this course is one of the contributions of this dissertation. We decided that the following learning objectives are most important to teach an engineering mindset [45] of software design.

Requirements analysis and specification are important skills for all software engineers [82, 6, 14, 141, 149], as prioritized requirements are the main drivers of software design [76, 134]. Therefore, a software design course should teach students how to elicit, specify, and prioritize requirements.

LO R (Requirements)	Bloom's Level [1]: **Analyzing**
Students should learn to: **Identify, describe, and prioritize relevant requirements for a given design problem.**	

Starting from requirements, design space exploration via constructive thinking and creative problem solving is the next required software design skill [45, 125]. Since considering multiple design alternatives is likely to lead to a better design [160], a software design course should teach students how to generate multiple viable solutions.

LO G (Generate)	Bloom's Level [1]: **Creating**
Students should learn to: **Generate multiple viable design solutions that appropriately satisfy the trade-offs between given requirements.**	

Modeling is a central aspect of design [54, 45, 137, 102] and essential for collaborative design [154, 86]. Hence, we should teach students to effectively communicate design ideas.

LO C (Communicate)	Bloom's Level [1]: **Creating**
Students should learn to: **Communicate the essential aspects of design solutions by choosing and visualizing appropriate abstractions and models.**	

Judging the quality of design options is essential to improve designs and assess requirements satisfaction [95]. Therefore, a software design course should teach design evaluation.

LO E (Evaluate)	Bloom's Level [1]: **Evaluating**
Students should learn to: **Evaluate design solutions based on their satisfaction of common design principles and trade-offs between different quality attributes.**	

Design decisions have a long-lasting impact on quality attributes, such as changeability, interoperability, reusability, robustness, scalability, and testability [155, 170, 173, 108]. To build on existing knowledge and experiences, teaching design principles can guide students to generate and evaluate design options for various quality attributes [138, 102].

LO DP (Design Principles)	Bloom's Level [1]: **Applying**
Students should learn to: **Describe, recognize, and apply principles for: Design for reuse, design with reuse, design for change, design for robustness, design for testability, design for interoperability, and design for scalability.**	

The software design process should be adjusted depending on the context, the overall amount of risk, and the types of risks in the domain [53]. Therefore, a software design course should teach students how to adjust the design process to fit into Agile, plan-driven, and risk-driven development processes across different domains.

LO P (Process)	Bloom's Level [1]: **Applying**
Students should learn to: **Determine and explain how to adapt a software design process to fit different development contexts and domains.**	

Date	Topic	LOs
L 1	Introduction and Motivation	
L 2	Problem vs. Solution Space	LO R, LO C
L 3	Design Abstractions	LO C
L 4	Quality Attributes and Trade-offs	LO R, LO E, LO C
L 5	Design Space Exploration	LO G
L 6	Generating Design Alternatives	LO G
L 7	Design for Change	LO DP, LO E, LO G
L 8	Design for Change	LO DP, LO E, LO G
L 9	Design for Interoperability	LO DP, LO C, LO E
L 10	Design for Testability	LO DP, LO E, LO G
L 11	Design with Reuse	LO DP, LO G, LO E
L 12	Reviewing Designs	LO E, LO C
	Midterm	
L 13	Cross-team Interface Design	LO MT
L 14	Design for Reuse	LO DP, LO E, LO G
L 15	Design for Scalability	LO DP, LO E, LO G
L 16	Design for Scalability	LO DP, LO E, LO G
L 17	Design for Robustness	LO DP, LO E, LO G
L 18	Design for Robustness	LO DP, LO E, LO G
L 19	Design Processes	LO P
L 20	Design for Security	LO DP, LO E, LO G
L 21	Design for Usability	LO DP, LO E, LO G
L 22	Ethical and Responsible Design	LO DP, LO E
L 23	Designing AI-based Systems	LO DP, LO E
L 24	Course Review	
	Project Presentations	LO C
	Final Exam	

Table B.1: Lecture topics and addressed learning objectives.

Finally, to build complex, large-scale software systems, skills of cross-team design and development are essential, as most modern software is built by more than one team [21, 22, 141, 154]. Thus, it is critical for a software design course to teach students how to collaborate across teams.

LO MT (Multi-Team) Bloom's Level [1]: **Creating**

Students should learn to: **Collaborate with other teams to design, develop, and integrate individually developed components into a complex system.**

B.4 Lecture Design

This section describes how the lectures in this course teach design primarily as an *activity* based on real-world case studies and constructivism learning theory. The list of lectures and learning objectives that they address is shown in Table B.1.

B.4.1 Teaching Design as an Activity via the *GCE-Paradigm*

We propose the "*GCE-paradigm*" as a guiding framework for systematically teaching software design activities. The GCE-paradigm describes software design as the process of iteratively ***g****enerating*, ***c****ommunicating*, and ***e****valuating* design options based on requirements. We introduce the GCE-paradigm via lectures and in-class activities on the individual design activities. Then, we teach how to combine these activities in an iterative design process while providing specific instruction on designing for individual quality attributes in "design for X" lectures. To help students connect new knowledge to the respective design activity, each slide highlights the associated activity in the cycle of the GCE-paradigm.

Requirements Analysis: To understand the problem and context of design tasks, we teach students to identify important requirements and domain assumptions (LO R). In Lecture 2, we illustrate the importance of domain assumptions based on the case study of the Lufthansa 2904 runway crash (caused by the assumption that the plane is on the ground if and only if the wheels are spinning, which was violated by a wet runway). We then ask students to identify important requirements and assumptions across different domains.

Communicating Designs via Abstractions: To support design collaboration and evaluation, we teach how to communicate designs using appropriate abstractions (LO C). Interleaved [55] throughout Lectures 2, 3, 4, and 9, we introduce context diagrams, component diagrams, sequence diagrams, data models, interface descriptions, and Class-Responsibility-Collaboration (CRC) cards. As a use of spaced repetition [91], we use these abstractions in following the lectures, recitations, homeworks, and project milestones.

Generating Design Alternatives: In Lecture 6, we survey techniques that help generate design options (LO G). First, we motivate the importance of thinking of different design alternatives, as this is likely to result in a better design [160]. Then, we teach brainstorming techniques (e.g., writing ideas on post-its, clustering, combining ideas, avoiding anchoring), which students practice during an in-class exercise. Based on the thereby introduced pattern of model-view-controller, we teach that design generation often starts with building on existing designs described in patterns.

Evaluating Design via Quality Attribute Trade-offs: As design often has to compromise between multiple conflicting objectives, we teach students how to identify and evaluate important quality attribute dimensions (LO E). In Lecture 4, we introduce quality attributes based on the connectors, publish-subscribe and call return, which can be used to implement the same functionality with different quality attributes. Thereby, we illustrate that design decisions can impact extensibility, robustness, and understandability. We then teach how to specify quality attribute requirements via measurable scenarios and show examples of trade-offs and synergies between quality attributes. In Lecture 12, we teach how to review designs via adversarial thinking and how to argue for design options. Via spaced repetition [91], we ask students throughout many lectures to identify important quality attribute dimensions, specify measurable scenarios, and evaluate design options.

Design Process: To convey the principle that the amount of design effort should depend on the criticality of the system being developed (LO P), we teach a risk-driven design approach [53] and show how this approach fits into Agile as well as more waterfall-like software development processes. Then, we conduct in-class activities to identify relevant risks for different domains (e.g., online shops, games, medical software, spacecraft systems, startups, and social media systems). Further, we teach the human aspects of software design [154, 161] by contrasting intuitive decision-making with rational decision-

making [139], discussing bounded rationality [90], and emphasizing that design is a collaborative hands-on activity [161].

Experience: At the end of the semester, we conducted an anonymous survey to request feedback on the course, including the lectures. 13 out of 17 students responded.

The students responded positively to the lectures. To the question "Which topics/lectures were valuable and should be kept for future versions of the course?" four students responded with "*all*" and two students responded with all "design for X" lectures. Lectures that students enjoyed in particular were the lectures on scalability (five students), reuse (three students), interoperability (two students), testability (two students), and changeability (two students). One student wrote: "*I think all the theoretical portion of the lectures were very well structured and should be all kept. Like this course is one of the best logically flowing courses I have taken at CMU.*"

No majority opinion emerged on which topics should be covered more/less. In response to the question "*To improve the course, which topics should we cover additionally, cover more, or cover less?*" students asked for more real-world examples in lectures (two students); more content on scalability (two students); and more content on testability, security, robustness, and quality attributes broadly (one student each).

Lesson Learned 1 (Design as an Activity) **Lectures**

Lectures on how to design large-scale software systems via the GCE-paradigm were well-received.

- Include a mix of lectures on individual design activities (requirements specification, design generation, design communication via abstractions, design evaluation, and design process adjustment) and on "design for X"
- To provide students with multiple practice opportunities, apply spaced repetition [91] by including the major activities in each "design for X" lecture while explicitly marking the corresponding slides with the activity name.

B.4.2 Real-World Case Studies

Case studies have been shown to be an effective teaching method in general software engineering education [148, 162, 60, 171] and have also been proposed for software design education in particular [37]. To convey the need for the design principles taught in the lectures (LO DP), we instructed them based on the following real-world case studies of well-known software failures and success stories, some of which we assigned as required readings before the corresponding lecture.

Global Distribution System In the lecture on *design for interoperability*, we used Global Distribution System[7] (the interface standard that is used by airlines and booking systems to transfer data between independently developed systems) as a case study for a multi-decade success of hundreds of interoperating systems (but with limited changeability).

Mars Climate Orbiter After discussing techniques to achieve syntactic interoperability, we used the Mars Climate Orbiter [24] case study to illustrate the importance of semantic interoperability (a mix of imperial units and metric units caused the system to crash for a multi-million dollar loss).

7 https://www.youtube.com/watch?v=1-m_Jjse-cs

Netflix's Simian Army: In the *design for testability* lecture, we used the Simian Army by Netflix as a positive example for quality attribute testing of large-scale systems [18].

Ariane 5 Rocket Launch Failure: In the *design with reuse* lecture, the well-known Ariane 5 failure (caused by an invalid assumption about the maximum velocity in the inertial reference system that was ported from Ariane 4) is used to illustrate the importance of identifying and checking assumptions made by reused components [110].

npm `left-pad`: In the *design with reuse* lecture, the suddenly unavailable, but widely reused npm package `left-pad`[8] with trivial implementation was used to motivate the design principle to strive for few dependencies.

Heartbleed: In the *design with reuse* lecture, the Heartbleed bug[9] (a security vulnerability in OpenSSL) motivated the importance of updating critical dependencies.

Twitter: In the *design for scalability* lecture, Twitter[10] (now X) was used as a case study to teach approaches for scaling a system based on estimated demand.

Experience: Overall, we believe the case studies were valuable for conveying the key course concepts and maintaining student engagement. We collected student feedback on the course in a mid-semester course feedback focus group session. To ensure students can speak freely and to anonymize all responses, the feedback was collected by an outside consultant who was not part of the course teaching team. In that session, all students unanimously agreed that the real-world case studies helped them learn, because "*examples of design scenarios and code snippets make core ideas more concrete and easier to understand*" and "*use of real-world examples in lecture[s] ties concepts to reality, helps retain info (e.g. the npm library)*". As instructors, we also noticed an increased level of student attention and participation specifically when discussing the case studies during lectures.

Lesson Learned 2 (Real-World Case Studies) **Lectures**

The use of real-world case studies of positive and negative examples for design principles has been well-received for teaching design principles (LO DP) and the software design process (LO P) in this course.

- For complex case studies, such as Global Distribution System and Netflix's Simian Army, assign required reading with a reading quiz before the lecture, so that all students are familiar with the important details of the case study.

B.4.3 Teaching Software Design Principles using Constructivism

In contrast to directly presenting design principles to students up-front, in this course, we let students themselves actively construct design principles by generalizing from real-world case studies of positive and negative examples (LO DP). Delivering lectures centered around student participation uses *active*

8 https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/
9 https://heartbleed.com/
10 https://blog.x.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how

learning [26], which has been shown to significantly improve learning outcomes in computer science and other fields [57, 73, 71]. Letting students construct design principles from examples is rooted in *constructivism learning theory*, which posits that teachers cannot simply transmit knowledge to students, but students need to actively construct knowledge in their own minds [15]. According to constructivism learning theory, students learn best by discovering information, checking new information against old information, and revising rules when they do not longer apply [15]. Based on the best available evidence in educational literature, constructivism improves retention [147], students' academic success [147], and meta-cognitive skills [128].

As software design principles are abstract concepts for which it is important to internalize why they exist and what their limitations are, we believe a constructivist teaching approach is most effective. By letting students follow the step-by-step process of formulating design principles based on positive and negative examples, we believe students gain a deeper understanding of how the design principles impact system design, why they often improve design, and in which cases they would not improve design.

For example, in the *design for interoperability* lecture, we use a case study based on Global Distribution System, a system that is used by nearly all airlines and booking systems to exchange data. First, we ask the students to discuss in small groups what specifically makes this example so successful and share their thoughts in the class. Second, we ask them to generalize their insights toward design principles that apply to future projects, which they described as creating a shared data format or an interface between systems. This is a part of the final design principle, but still missing an important element. Hence, we show the students the example of the Mars Climate Orbiter failure [24] (which resulted from the inconsistent use of metric and imperial units) to demonstrate that just having syntactic interoperability alone is not sufficient, but that semantics have to be defined precisely as well. Students appropriately inferred the design principle of documenting the meaning and units of interfaces. Finally, we let students describe the shortcomings of Global Distribution System. They correctly identified limited changeability of the interface that is implemented across hundreds of systems. In doing so, students identified and addressed the concrete challenges, generalized them, and constructed the design principles that the lecture was intended to teach. We follow the same approach to teaching design principles throughout the course.

To identify whether non-participating students also understood the design principles, we end each lecture with an *exit ticket* [56] (a digital assignment in which students are asked to summarize the lecture's main message in their own words and apply it to a small, different example).

Experience: In the mid-semester focus group, 46 % of students agreed that in-class discussions helped them learn, since "*in class discussions help us think and reason over content*" and facilitate "*reiteration of ideas; students have different perspectives*". Considering that students often subjectively under-value the objective effectiveness of active learning techniques [49], these results suggest that constructivism likely supported the students' learning of design principles. Based on the quote "*[we] don't know what they expect as answers when they put us into discussion groups*", we identify the clarity of questions as a potential challenge of the technique, as students might not have always known what type of answer was expected of them. Finally, all students agreed that exit tickets helped them learn, because "*exit tickets help us reconsider what we learned in the class right after class*".

Lesson Learned 3 (Constructivism) **Lectures**

The use of constructivism for teaching design principles (LO DP) was overall well-received in this course.

- Give students 2 – 5 min of silent thinking and small-group discussions before discussing with the whole class.
- Soon after describing design principles, give students another problem to practice applying the principles in recitations or homework.
- To give students an idea of what type of answer is expected, give them examples of answers to a similar question that they are already familiar with.
- At the end of each lecture, include an exit ticket with one summary task and one small task for applying the learned techniques to a different example.

B.5 Multi-Team Project

While teamwork is one of the most important soft skills in professional software development [6], graduates in computer science often lack the skill to collaborate across teams [141, 22] or work on large projects [141]. To let students practice collaborative software design in a realistic context, in which no single developer fully understands all components, we decided to include a large-scale multi-team project in this course (LO MT). In the project, each team developed its own medical appointment scheduling app and one of four collaborating services. The *medical scheduling app* should allow users to book appointment slots, see their results, and receive quarantine requests. The *healthcare administrator service* should let healthcare professionals enter patients' test results and other medical data. The *policymaker service* should allow government officials to modify the policy that determines whether and for how long a patient should undergo quarantine. The *central database service* provides storage and retrieval of patient information across multiple scheduling apps. The *public information service* should allow users to view aggregated statistics). The teams were eventually asked to integrate their scheduling app and service with other teams' services.

The decision to let students collaboratively design and develop a large-scale system comes with unique challenges that should be addressed by course design to ensure students focus their time and effort on the main learning objectives and can gain a mostly positive experience with the design techniques. These major challenges include:

- Challenges of cross-team communication [107], which we address by letting teams pick a dedicated member to be responsible for cross-team communication (Section B.5.1)
- Potentially incompatible interfaces of individually developed services, which we address using interface descriptions (Section B.5.2)
- Challenges of testing services while dependent-on services have not been implemented, which we address using test double components (Section B.5.3)

To better support students with the project, we offered weekly project office hours (15 min slots per

team) during which students could present their progress, ask clarification questions, and receive targeted feedback from instructors.

Experience: Students particularly valued the weekly project office hours, with quotes such as "*I really gained a lot from your feedback and discussion with you during the office hours. It enhanced my learning and thinking about previous or undergoing milestones.*". The four teams built a system with a total size of 19.5 KLOC. This amounts to 1.15 KLOC per student on average. Overall, the developed system was functionally correct, and services integrated well with each other. The course project provided many insightful learning opportunities, which are discussed in the following sections.

B.5.1 Cross-Team Communicator

As identified in previous work on multi-project software engineering courses [107, 43, 42], communication between teams is a major challenge. To reduce communication overhead between teams (LO MT) we decided to use class time for cross-team communication, provided a shared Slack channel for cross-team communication, and dedicated a *cross-team communicator* role for each team. Cross-team communicators should serve as interfaces of the team and represent the wishes and needs of their team. When multiple teams need to make decisions together, instead of all students meeting, discussions can be limited to only cross-team communicators.

Experience: We believe some teams did not pick the ideal person to serve as the cross-team communicator. During the initial design of the high-level architecture, cross-team communicators met to assign component responsibilities. As some teams picked students who were less involved in the team's technical design discussions as cross-team communicators, they did not fully understand the technical implications of these decisions on the team's workload and required technical expertise. This led to unpleasant surprises when the students learned that their cross-team communicator agreed to them working on tasks that they did not feel equipped to work on in the given time frame, requiring a new meeting to redesign the system's overall architecture.

Lesson Learned 4 (Cross-Team Communicator) **Project**

The effectiveness of cross-team communicators depends on how well they can evaluate design trade-offs and how well they know the skill set of their team.

- To reduce the risks of multi-team challenges (LO MT), let teams pick a cross-team communication that will serve as a facade of the team and interface with other teams.
- Clearly describe the responsibilities and desired traits of a cross-team communicator.
- Ensure that cross-team communicator is not a role that teams assign to the member who has not contributed enough yet, but a role that should be given to a student who is prepared to represent the team's needs and wishes in important technical design decisions.

B.5.2 Service Interface Description

To give students the experience of building a component that is used by other teams and using components developed by other teams (LO MT), we let teams describe OpenAPI specifications describing syntax and semantics of their interfaces (LO C) and review each others' interfaces (LO E).

Experience: Students had only a few integration issues. Considering that each service was developed individually and most students experienced a large-scale development project with multiple teams for the first time, we were surprised by the high interface compatibility between the services. We believe interface descriptions contributed to this success.

Lesson Learned 5 (Interface Descriptions) **Project**

Interface descriptions likely helped students independently develop compatible services (LO MT).

- As part of the project milestone in which teams design their individual services (Milestone 3), include a task for students to precisely specify interface descriptions.
- To increase the probability of major compatibility issues being caught before implementation, ask student teams to give each other feedback on their interface descriptions.

B.5.3 Test Double Components

While all teams develop their own services, dependent-on-services are not immediately available for testing. To address this challenge and to allow students to simulate data sent from other components (LO MT), we taught students to implement *test double components* (components that mimic the interface of a required service to control indirect inputs or verify indirect outputs [126]) based on interface specifications in the *design for testability* lecture. During the project, we asked students to implement test doubles for dependent-on components.

Experience: Test doubles helped students find some, but not all, bugs before integration. Students also mentioned that in the project, test double components helped "*isolating the influence of external components*". Many teams implemented test doubles via conditional logic within their components, rather than as a separate HTTP-communicating component, which impeded replacing them with real components later.

Lesson Learned 6 (Test Double Components) **Project**

Test double components helped students independently develop and integrate services (LO MT).

- To ease replacing test double components with the real components, recommend students to implement test double components by mocking HTTP messages rather than simply mocking functions inside their own component.
- To simplify implementation tasks, point students to libraries and frameworks that inject HTTP messages.

B.5.4 Milestone Reports

Many companies, such as Google, use Design Docs or other architecture decision records [5] to describe their important design decisions [174]. Students practiced writing similar documents in milestone reports

for which we asked them to generate (LO G), communicate (LO C), and evaluate (LO E) multiple design options for project tasks. The following sections describe each milestone and our experience.

Milestone 1 (Domain Modeling & Initial System Design)

In the first milestone, students were given the description of a small design problem (designing a medical appointment scheduling app). Based on the given requirements and context, students were asked to model a problem domain (LO C), identify important quality attribute requirements (LO R), and describe a first high-level design solution (LO G and LO C).

Experience: In an end-of-semester survey asking for feedback on every milestone, virtually all students said this milestone was "*Good*" or "*Great*" and spent less time on the milestone than we anticipated. Based on the submitted reports, students made fewer design decisions (especially on the choice of technologies and web frameworks) than we anticipated. Therefore, we recommend including more mandatory questions on particularly important decisions so that more design decisions are made in this milestone.

Milestone 2 (First Prototype Development)

In the second milestone, students should refine (LO G), model (LO C), and implement the design they described in Milestone 1, implement tests to evaluate the end-to-end functionality (LO E), and reflect on how the design changed and which other alternatives options they considered (LO G).

Experience: Students took more time for this milestone than we anticipated, requiring us to extend the milestone by one week. In the end-of-semester survey, many students said "*More time should be given to this milestone because ... some of the members in the group are still in the learning stage of some frontend/backend framework.*". Furthermore, due to the higher workload of picking and learning a framework, students' time efforts shifted more towards implementation than design, leaving less time to consider alternatives and evaluate the impact of implementation decisions on the system design [108]. Providing more implementation support, specifically on frameworks that might be useful for the project, might help address this issue.

Lesson Learned 7 (Implementation Support) — **Project**

The relative portion of project time spent on coding rather than design was higher than desired, resulting in students investing less time into the main LOs.

- To reduce the time students spend on coding and allow them to focus more on design activities, include coding templates that help students implement their systems more efficiently.
- Link tutorials to common frameworks and libraries.
- Include a recitation at the beginning of the course that introduces commonly used code generation techniques.

Milestone 3 (Design for Changeability & Interoperability)

In the third milestone, students were first introduced to the four services that they were going to design and implement to interoperate with each other. The milestone provides a description of the functionality

of each service as well as tips for cross-team collaboration via cross-team channels and a dedicated cross-team communicator. Based on this description and service assignment per team, students are asked to design their service (LO G), model it using interface descriptions (LO C), and collaborate with other teams to ensure compatibility (LO MT). To further support service compatibility, students are asked to design test doubles for two of the most central services. Students are also asked to re-design their appointment scheduling app to support certain future changes (LO G) and add tests to evaluate the functionality (LO E). In a design reflection students should report on design decisions they made during interface design, the changes they made and describe a change impact analysis of two potential changes.

Experience: Students had major discussions and disagreements, which increased the workload of the milestone while providing insightful learning opportunities. We recommend providing multiple opportunities for students to have cross-team discussions in recitations or setting some lecture time aside for this, as some students mentioned they had "*not enough time to discuss design decisions with other students*".

Milestone 4 (Service Development & Integration)

In the first part of the fourth milestone, we asked teams to implement their services, while collaborating with other teams to ensure compatibility (LO MT), and implement test doubles for adjacent services. Then they should deploy their services and provide other teams with the URL and port of their service instance. In the second part, students should integrate their services by replacing the test double components with the real deployed services of other teams. Then they should perform rigorous integration testing to evaluate the functionality of the overall system (LO E). In a design reflection students should report on the design principles they used (LO DP), how they reused existing libraries, how cross-team collaboration affected their design decisions, and how starting from a fixed interface impacted their implementation.

Experience: The integration of services went largely smoothly. The most common integration issues were related to different capitalization and the use of dashes in data formats that resulted from interface changes that were not explicitly communicated but were easy to fix. In the end-of-semester survey, students mentioned this milestone "*helped understand teamwork and how to collaboratively work with others*".

Milestone 5 (Robustness Testing)

In the last milestone each team is assigned the service of another team for which they should conduct intense robustness testing by trying to break the service (LO E). They should report their findings to the team that developed the service. In an optional task, students were asked to describe at least two design options for at least two of the issues found by other teams and describe the improved designs (LO G and LO C). Due to time limitations and due to this task strongly relying on the findings of other teams this task only gave bonus points. However, all teams completed this optional task.

Experience: Students thoroughly enjoyed breaking the services of other teams and said it was "*useful to understand what issues a system can potentially face and what could be potential loopholes*". As students spend less time on this than we expected, expanding the milestone by asking the students to identify a large variety of issues (e.g., performance, correctness, availability, security) is one potential improvement.

B.5.5 Assessment of Milestone Report Submissions

Asking students to submit multiple written reports on the progress of their project lets students receive constructive feedback and observe their own growth [70]. The main shortcomings of submissions were related to LO G and LO C.

The discussion of design alternatives was often quite superficial. In some cases, students just described their final design without discussing potential alternatives. In other cases, students described alternative designs that clearly would not satisfy the requirements and thereby missed the opportunities to meaningfully discuss design trade-offs.

The models of design solutions often did not communicate the essential aspects of the corresponding design. In many cases, the textual arguments of students were largely disconnected from the presented diagrams, suggesting that students did not sufficiently consider what aspects of their design should be communicated at which level of detail. In other cases, models were too ambiguous or unclear.

We allowed students to redo some milestones to improve their design discussions. We saw significant growth in redone milestones, later milestones, and during final presentations, suggesting that feedback helped students improve.

Lesson Learned 8 (Milestone Reports) **Project**

Milestone reports have helped assess students' progress and their satisfaction of learning objectives and have been great opportunities to provide targeted feedback to teams in this course.

- To allow students to apply feedback in the next milestone, try to grade submissions quickly.
- Allow students to redo some milestone reports for an improved grade to incentivize students to take provided feedback seriously.

B.6 Homework Assignments

This section describes our design and experience of complementing the project with individual homework assignments.

B.6.1 HW1 - Domain and Design Modeling

The first homework is designed to let students practice domain analysis (LO R) and modeling (LO C). The homework is scheduled so that students receive feedback on this homework before working on the first project milestone.

In the homework, students were presented with a case study of a home security system and asked to model the system using a context model, component diagram, data model, and sequence diagram. Students should also describe assumptions made about the domain and design decisions they made.

Experience: In an end-of-semester survey students overall liked the homework while mentioning a higher-than-expected workload (e.g., "*This was useful and a must learn skill for design documentation. Although it took me around 6 – 7 hours as opposed to 2 – 3 hours.*"). Most submissions demonstrated accomplishment of the learning objectives. The most common mistake was that 18 % of submissions included domain entities in component diagrams rather than context diagrams.

B.6.2 HW2 - Design for Reuse

The second homework practiced generating multiple design alternatives (LO G), communicating them using interface descriptions (LO C), evaluating them for reusability (LO E), and describing the design principles they support (LO DP).

Students were tasked to evaluate an open-source package for reusability by identifying its assumptions and reuse context, describing design principles that contribute to its reusability, and describing reuse scenarios in which reusing it would be appropriate and inappropriate. Then, students were asked to improve the package design for an unsatisfied reuse scenario and communicate the new design with interface descriptions and a description of required implementation changes. Finally, students should describe how the redesign improves the reusability based on applied design principles or other arguments. The homework was designed to be open-ended to allow students to freely explore the reusability of the given module based on their interests and domain expertise.

Experience: In the end-of-semester survey, students overall liked the homework (e.g., "*Very good. Required much more thought about the reuse and how it works in practice.*"). Three students mentioned that "*the instruction was very open-ended*", suggesting that some students prefer more concrete instructions rather than an open-ended format.

In the graded submissions, most students demonstrated sufficient accomplishment of the learning objectives. The most common mistakes were related to the precise description of reuse scenarios (35 % of submissions), and partially lacking description of semantics in the interfaces (6 % of submissions).

B.6.3 HW3 - Design for Scalability

The third homework was designed to provide students with design generation (LO G), communication (LO C), and evaluation (LO E) skills related to scalability. Based on the case study of the project, students should specify scalability requirements, make design decisions (e.g, what data to store, what storage model to use, what type of scaling to use, how to distribute the data, which data to cache), model them using component diagrams, and evaluate the designs.

Experience: In the end-of-semester survey all students liked the homework (e.g., "*It was a good balance between the time spend and learning outcome*").

In the graded submissions, almost all students demonstrated sufficient accomplishment of the learning objectives. Common mistakes were mostly minor, such as the use of generic rather than domain-specific component names, insufficient justifications of design decisions, and unrealistic demand estimations.

B.7 Open Challenges of Teaching Design

The main goal of this course was to teach students how to design large-scale software systems by fostering an engineering mindset and teaching design as an activity. Overall, students struggled most with learning objectives LO G, LO C, and LO MT. As all three LOs are at the highest cognitive level of Bloom's revised taxonomy [1] (Creating), they are particularly challenging to teach effectively. In this section, we discuss the concrete challenges we observed and suggest ideas to overcome them in future courses.

B.7.1 Generating Multiple Viable Alternatives

As mentioned in Section B.5.5, in milestone reports, students struggled with generating multiple viable alternative design options (LO G). We observed similar trends in both exams (mid-term and final exam), in which we asked students to describe at least two viable design options for a design problem, evaluate them, and discuss trade-offs between the two options. In both exams, especially in the mid-term, many students presented one viable option and one straw-man option that was a deliberate degradation of their other option.

As generating multiple viable design options is an important software design skill [160], we see this as an important challenge when teaching design. While students' ability to discuss alternatives noticeably improved throughout the course, we believe providing more dedicated instruction on design generation is still an open challenge. Potential improvements could teach more design generation and brainstorming techniques throughout the course paired with exercises of generating as many viable ideas as possible to give students more practice and spaced repetition. Furthermore, as students asked for "*more concrete tactics*" to design systems, a curated list of more specific design recipes, cautiously annotated with limitations of their applicability, could help students learn the generation of more design options.

Lesson Learned 9 (Multiple Viable Alternatives)

Many students in this course struggled with describing multiple, viable design alternatives.

- Include multiple individual homeworks, recitations, and in-class exercises for students to practice generating multiple design alternatives.
- Teach more concrete guidelines on how to generate multiple viable design alternatives.

B.7.2 Design Communication via Appropriate Abstractions

As mentioned in Section B.5.5, in milestone reports, students struggled with identifying appropriate abstractions to communicate the essential aspects of their design (LO C). We observed similar trends in both exams, in which we asked students to communicate designs using component diagrams, interface diagrams, and sequence diagrams.

In the mid-term exam, students struggled most severely with interface descriptions and component diagrams. Only 58 % of submissions demonstrated sufficient accomplishment of the learning objective (6 % did not include an answer to the question, 12 % did not describe interfaces using an appropriate format, and 24 % lacked descriptions of semantics). Interface descriptions improved in the final exam with 82 % of submissions demonstrating sufficient accomplishment of the learning objective. The improvement is most likely due to students having had more practice with interface descriptions in the project and Homework 3. Therefore, we believe adding additional homework to practice interface descriptions in the first half of the semester would help students. The additional homework workload might be offset via Lesson Learned 7. Common mistakes for component diagrams included unclear responsibility assignments, missing arrows, and missing connection labels. Mid-term submissions included more severe cases of diagrams being too ambiguous to appropriately convey design choices, suggesting some growth. Furthermore, in both exams, some diagrams were inconsistent (i.e., design choices communicated in different models contradicted each other).

Based on these observations, we identified that teaching the identification of appropriate abstraction to model the most essential aspects of design is still an open challenge. Potential improvements could use interleaving [55] of different model types to train students to identify which aspects of a design are best represented using which type of model. Many exercises in modeling different design aspects throughout the course could give students more practice and spaced repetition.

Lesson Learned 10 (Communicate Abstractions)

Many students in this course struggled with communicating design options via appropriate abstractions.

- Include multiple opportunities for students to practice interface descriptions and component diagrams in individual homeworks, recitations, and in-class exercises.
- Include guidelines and exercises on selecting abstractions that communicate the essential aspects of a given design.

B.7.3 Cross-Team Design Debate

One major challenge during the multi-team project was how to design the system in a way that the implementation effort of each service is roughly equal (LO MT). Three teams devised a design that would assign major responsibilities to the central database, whose team was largely absent during these discussions. Understandably, the database team was opposed to taking on a higher workload. Faced with this conflict in a situation in which the three other teams invested considerable effort into a design that was not going to get approved by the other team, a heated discussion took place on Slack. To lead students toward a more constructive resolution, we recommended an in-person meeting. With instructors only passively observing, the teams self-organized a collaborative discussion of potential design options and evaluated them across self-identified dimensions (code modifications needed, interface complexity, extensibility, and workload balance). Based on their evaluations, teams then voted for their preferred option and democratically reached a reasonable consensus.

While this discussion initially resulted from frustrations and disagreements between teams, it provided one of the best learning opportunities to experience the complexity of real-world design considerations [154, 161]. During this meeting, students demonstrated excellent application of advanced software design skills, such as trade-off evaluation, design communication, iterative refinement, and a deep understanding of the non-technical implications of their decisions, skills that we did not observe in the students before. We believe this discussion particularly helped students grow and integrate all major design skills more than they would have otherwise.

Therefore, we recommend explicitly integrating more opportunities for student teams to collectively debate cross-team decisions. While we allocated one lecture at the beginning of Milestone 3 for this activity, due to most students of the database team not attending, and students having had little time to generate design alternatives before this discussion, it was less productive than the debate following the heated Slack discussion. A challenge in integrating cross-team debates is to identify the right balance between leaving enough opportunities for constructive disagreements between teams to encourage debates while moderating the discussions enough to ensure that students still have a positive experience.

Lesson Learned 11 (Design Debates)

Students gained the most substantial practice with multi-team software design activities during an unplanned cross-team design debate.

- Include multiple opportunities for teams to debate cross-team design decisions during recitations or lectures.
- Embrace (constructive) disagreements between teams as an opportunity to practice group decision-making.
- While avoiding too much interference with student autonomy, ensure that disagreements are resolved peacefully.

B.8 Conclusions and Implications for the Dissertation

This chapter has presented an educational approach to prevent bugs that result from incorrect composition of software components. It described the design of a novel course on designing large-scale software systems via the GCE-paradigm using real-world case studies, constructivism, and a multi-team project. This approach complements automated bug-finding techniques and visual diagramming approaches, as it intended to amplify the benefits of such techniques by fostering a an architect's mindset. Our experience motivates future work that empirically measures whether students with this type of course instruction produce fewer architecture misconfiguration bugs in their future projects and/or identify architecture misconfiguration bugs more effectively would provide further support for the hypothesis that this education provides students with the skills to prevent and identify architecture misconfiguration bugs.

Glossary

A

API: An Application Programming Interface (API) is a set of specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API.
Pages: iii, 4, 5, 9, 14, 21, 22, 25, 34, 47, 49, 63, 66

AST: An Abstract Syntax Tree (AST) is a representation of source code that hierarchically splits programing language constructs into parent-child relationships.
Pages: 22

B

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

Behavioral view: The behavioral view, also known as dynamic view, expresses the behavior of the system or its parts. It can describe input-output relationships for components, states, state transitions, actions, and other behavioral properties such as timing. See Section 2.1.3.
Pages: 3, 8, 11, 103

C

CI: Continuous Integration (CI) is the automated build and testing stages of the software release process.
Pages: 65

Component: A component is an independently deployable run-time unit of software (e.g., processes) that communicates with other components primarily via messages. See Section 2.1.2.
Pages: 7–9, 11, 12, 21–30, 33, 51–53, 55, 60, 63, 65–68, 103, 104

Component-connector model: A component-connector model is an instance of the component-connector view. See Section 2.1.2.
Pages: 3, 11–13, 51

Component-connector view: The component-connector view, also known as run-time architecture, represents a structural configuration of the architecture at run time containing components, connectors, and ports. See Section 2.1.2.
Pages: 7, 11, 103

Component-port-connector diagram: A component-port-connector diagram is an instance of a component-connector view that also shows ports.
Pages: 51

Connector: A connector is an architectural element that specifies the mechanisms by which components communicate, coordinate, and transfer control or data. See Section 2.1.2.
Pages: 8, 103

Constant-time sleep call: A constant-time sleep call sleeps the thread for the same amount of time every time it is are called. See Section 2.4.4 for examples.
Pages: 11

CUDA: Compute Unified Device Architecture (CUDA) is a Graphics Processing Unit (GPU)-based parallel computing framework and API.
Pages: 47

F

Filling-time sleep call: A filling-time sleep calls sleeps the thread for the remainder of a periodic interval every time they are called. See Section 2.4.4 for examples.
Pages: 11

G

Global Distribution System: The A Global Distribution System (GDS) is a network that connects digital systems for flights, hotels, and car rentals.
Pages: 89–91

GPU: Graphics Processing Unit.
Pages: 104

L

LLM: A large language model (LLM) is a type of artificial intelligence model, specifically a deep learning model, that excels at understanding, generating, and manipulating human language.
Pages: 49, 67

LTL: Linear Temporal Logic.
Pages: 12, 16

M

Module: A module is a a unit of code.
Pages: 9, 104

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

N

Node: A node is a component in ROS. See Section 2.4.2.
Pages: 10, 11, 17, 47

P

Package: A package is a module in ROS. See Section 2.4.1.
Pages: 9

Port: A port identifies a specific point where a component interacts with its environment via inputs (input port) or outouts (output port). See Section 2.1.2.
Pages: 3, 5, 10, 11, 103

Publish-subscribe: Publish-Subscribe is an asynchronous message sending connector that loosely couples senders (i.e., *publishers*) from receivers (i.e., *subscribers*) via a know intermediary interface. See Section 2.2.1.
Pages: 8, 10, 11, 105

Publisher: A publisher is the role of the sender in the publish-subscribe style. It sends messages to topics that forward them to every component that subscribed before the message was sent. See Section 2.2.1.
Pages: 8, 10, 47, 105

R

ROS: The Robot Operating System (ROS) is the most popular open-source framework for component-based robotics systems. See Section 2.4.
Pages: iii, 3–6, 9–11, 14, 16, 17, 21, 25, 34, 36, 46, 47, 49, 51, 63, 66

S

Software engineering: Software engineering is a branch of computer science that creates cost-effective solutions to practical computing problems by applying codified knowledge for developing software systems in the service of mankind.
Pages: 83, 84, 89, 93

Subscriber: A subscriber is the role of the receiver in the publish-subscribe style. It after it subscribed to a topics it receives all messages sent by the corresponding publishers. See Section 2.2.1.
Pages: 8, 10, 47, 105

T

Topic: Topics is ROS implementations of the publish-subscribe style. They are represented as strings. See Section 2.4.3.
Pages: 3, 8, 21, 47, 105

U

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

Bibliography

[1] ACM Committee for Computing Education in Community Colleges (CCECC). 2023. **Bloom's for Computing: Enhancing Bloom's Revised Taxonomy with Verbs for Computing Disciplines**. DOI: 10.1145/3587276.

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

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

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

[5] Bardha Ahmeti, Maja Linder, Raffaela Groner, and Rebekka Wohlrab. 2024. **Architecture Decision Records in Practice: An Action Research Study**. In *Software Architecture*. Springer Nature Switzerland, 333–349. DOI: 10.1007/978-3-031-70797-1_22.

[6] Deniz Akdur. 2022. **Analysis of Software Engineering Skills Gap in the Industry**. *ACM Trans. Comput. Educ.*, 23, 1, Article 16, (December 2022), 28 pages. DOI: 10.1145/3567837.

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

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

[9] Robert Allen and David Garlan. 1994. **Formalizing architectural connection**. In *International Conference on Software Engineering*, 71–80. DOI: 10.1109/ICSE.1994.296767.

[10] 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.

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

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

[13] Jocelyn Armarego. 2002. **Advanced Software Design: a Case in Problem-based Learning**. In *Conference on Software Engineering Education and Training* (CSEE&T '02), 44–54. DOI: 10.1109/CSEE.2002.995197.

[14] Nana Assyne, Hadi Ghanbari, and Mirja Pulkkinen. 2022. **The state of research on software engineering competencies: A systematic mapping study**. *Journal of Systems and Software*, 185, 111183. DOI: 10.1016/j.jss.2021.111183.

[15] Steve Olusegun Bada and Steve Olusegun. 2015. **Constructivism Learning Theory: A Paradigm for Teaching and Learning**. *Journal of Research & Method in Education* (IOSR-JRME), 5, 6, 66–70. https://iosrjournals.org/iosr-jrme/papers/Vol-5%20Issue-6/Version-1/I05616670.pdf.

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

[17] Alex Baker and André van der Hoek. 2009. **An Experience Report on the Design and Delivery of Two New Software Design Courses**. In *Technical Symposium on Computer Science Education* (SIGCSE '09). ACM, 519–523. DOI: 10.1145/1508865.1509045.

[18] Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. 2019. **Automating Chaos Experiments in Production**. In *International Conference on Software Engineering: Software Engineering in Practice* (ICSE-SEIP '19), 31–40. DOI: 10.1109/ICSE-SEIP.2019.00012.

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

[20] 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.

[21] Andrew Begel, Nachiappan Nagappan, Christopher Poile, and Lucas Layman. 2009. **Coordination in Large-Scale Software Teams**. In *ICSE Workshop on Cooperative and Human Aspects on Software Engineering* (CHASE '09). IEEE, 1–7. DOI: 10.1109/CHASE.2009.5071401.

[22] Andrew Begel and Beth Simon. 2008. **Novice Software Developers, All Over Again**. In *International Workshop on Computing Education Research* (ICER '08). ACM, 3–14. DOI: 10.1145/1404520.1404522.

[23] 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.

[24] Mishap Investigation Board. 1999. Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999. (1999). https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf.

[25] 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/.

[26] Charles C Bonwell and James A Eison. 1991. **Active Learning: Creating Excitement in theClassroom. 1991 ASHE-ERIC Higher EducationReports**. ERIC. https://eric.ed.gov/?id=ED336049.

[27] 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.

[28] 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.

[29] 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.

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

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

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

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

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

[35] Tamara Carleton and Larry Leifer. 2009. **Stanford's ME310 Course as an Evolution of Engineering Design**. In *CIRP Design Conference – Competitive Design*. Cranfield University Press. http://hdl.handle.net/1826/3648.

[36] D. Carrington and S.-K. Kim. 2003. **Teaching Software Design with Open Source Software**. In *Frontiers in Education* (FIE '03). Volume 3, S1C–9. DOI: 10.1109/FIE.2003.1265910.

[37] Chun Yong Chong, Eunsuk Kang, and Mary Shaw. 2023. **Open Design Case Study - A Crowdsourcing Effort to Curate Software Design Case Studies**. In *International Conference on Software Engineering: Software Engineering Education and Training* (ICSE-SEET '23), 23–28. DOI: 10.1109/ICSE-SEET58685.2023.00008.

[38] 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.

[39] Tony Clear, Sarah Beecham, John Barr, Mats Daniels, Roger McDermott, Michael Oudshoorn, Airina Savickaite, and John Noll. 2015. **Challenges and Recommendations for the Design and Conduct of Global Software Engineering Courses: A Systematic Review**. In *ITiCSE on Working Group Reports* (ITICSE-WGR '15). ACM, 1–39. DOI: 10.1145/2858796.2858797.

[40] 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.

[41] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. **srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration**. In *International Conference on Software Maintenance*, 516–519. DOI: 10.1109/ICSM.2013.85.

[42] David Coppit. 2006. **Implementing Large Projects in Software Engineering Courses**. *Computer Science Education*, 16, 1, 53–73. DOI: 10.1080/08993400600600443.

[43] David Coppit and Jennifer M. Haddox-Schatz. 2005. **Large Team Projects in Software Engineering Courses**. In *Technical Symposium on Computer Science Education* (SIGCSE '05). ACM, 137–141. DOI: 10.1145/1047344.1047400.

[44] 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.

[45] Nigel Cross. 1982. **Designerly ways of knowing**. *Design Studies*, 3, 4, 221–227. Special Issue Design Education. DOI: 10.1016/0142-694X(82)90040-0.

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

[47] Daniela Damian, Allyson Hadwin, and Ban Al-Ani. 2006. **An Experiment on Teaching Coordination in a Globally Distributed Software Engineering Class**. In *International Conference on Software Engineering* (ICSE '06). ACM, 685–690. DOI: 10.1145/1134285.1134391.

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

[49] Louis Deslauriers, Logan S. McCarty, Kelly Miller, Kristina Callaghan, and Greg Kestin. 2019. **Measuring actual learning versus feeling of learning in response to being actively engaged in the classroom**. *Proceedings of the National Academy of Sciences*, 116, 39, 19251–19257. DOI: 10.1073/pnas.1821936116.

[50] 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.

[51] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. 2006. **Can Graduating Students Design Software Systems?** In *Technical Symposium on Computer Science Education* (SIGCSE '06). ACM, 403–407. DOI: 10.1145/1121341.1121468.

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

[53] George Fairbanks. 2010. **Just Enough Software Architecture: A Risk-Driven Approach**. Marshall & Brainerd.

[54] George Fairbanks. 2023. **Software Architecture is a Set of Abstractions**. *IEEE Software*, 40, 4, 110–113. DOI: 10.1109/MS.2023.3269675.

[55] Jonathan Firth, Ian Rivers, and James Boyle. 2021. **A systematic review of interleaving as a concept learning strategy**. *Review of Education*, 9, 2, 642–684. DOI: 10.1002/rev3.3266.

[56] Kelsie Fowler, Mark Windschitl, and Jennifer Richards. 2019. **Exit Tickets**. *The Science Teacher*, 86, 8, 18–26. DOI: 10.1080/00368555.2019.12293416.

[57] Scott Freeman, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Hannah Jordt, and Mary Pat Wenderoth. 2014. **Active learning increases student performance in science, engineering, and mathematics**. *Proceedings of the National Academy of Sciences*, 111, 23, 8410–8415. DOI: 10.1073/pnas.1319030111.

[58] Matthias Galster and Samuil Angelov. 2016. **What makes teaching software architecture difficult?** In *International Conference on Software Engineering Companion* (ICSE '16). ACM, 356–359. DOI: 10.1145/2889160.2889187.

[59] 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.

[60] Kirti Garg and Vasudeva Varma. 2007. **A Study of the Effectiveness of Case Study Approach in Software Engineering Education**. In *Conference on Software Engineering Education and Training* (CSEE&T '07), 309–316. DOI: 10.1109/CSEET.2007.8.

[61] D. Garlan, R. Allen, and J. Ockerbloom. 1995. **Architectural Mismatch: Why Reuse is so Hard**. *IEEE Software*, 12, 6, 17–26. DOI: 10.1109/52.469757.

[62] David Garlan, Mary Shaw, Chris Okasaki, Curtis M. Scott, and Roy F. Swonger. 1992. **Experience with a Course on Architectures for Software Systems**. In *Software Engineering Education*. Springer Berlin Heidelberg, 23–43. DOI: 10.1007/3-540-55963-9_38.

[63] Vahid Garousi, Görkem Giray, Eray Tüzün, Cagatay Catal, and Michael Felderer. 2019. **Aligning software engineering education with industrial needs: A meta-analysis**. *Journal of Systems and Software*, 156, 65–83. DOI: 10.1016/j.jss.2019.06.044.

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

[65] Carlo Ghezzi and Dino Mandrioli. 2006. **The Challenges of Software Engineering Education**. In *Software Engineering Education in the Modern Age*. Springer Berlin Heidelberg, 115–127. DOI: 10.1007/11949374_8.

[66] 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.

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

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

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

[70] Randall S. Hansen. 2006. **Benefits and Problems With Student Teams: Suggestions for Improving Team Projects**. *Journal of Education for Business*, 82, 1, 11–19. DOI: 10.3200/JOEB.82.1.11-19.

[71] Qiang Hao, Bradley Barnes, Ewan Wright, and Eunjung Kim. 2018. **Effects of Active Learning Environments and Instructional Methods in Computer Science Education**. In *Technical Symposium on Computer Science Education* (SIGCSE '18). ACM, 934–939. DOI: 10.1145/3159450.3159451.

[72] 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.

[73] Susanna Hartikainen, Heta Rintala, Laura Pylväs, and Petri Nokelainen. 2019. **The Concept of Active Learning and the Measurement of Learning Outcomes: A Review of Research in Engineering Higher Education**. *Education Sciences*, 9, 4. DOI: 10.3390/educsci9040276.

[74] 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.

[75] Rune Hjelsvold and Deepti Mishra. 2019. **Exploring and Expanding GSE Education with Open Source Software Development**. *ACM Trans. Comput. Educ.*, 19, 2, Article 12, (January 2019), 23 pages. DOI: 10.1145/3230012.

[76] Chenglie Hu. 2013. **The nature of software design and its teaching: an exposition**. *ACM Inroads*, 4, 2, (June 2013), 62–72. DOI: 10.1145/2465085.2465103.

[77] 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.

[78] 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.

[79] 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.

[80] ISO/IEC/IEEE. 2011. **ISO/IEC/IEEE Systems and Software Engineering – Architecture Description**. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, 1–46. DOI: 10.1109/IEEESTD.2011.6129467.

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

[82] Michael Jackson. 1995. **The World and the Machine**. In *International Conference on Software Engineering* (ICSE '95). ACM, 283–292. DOI: 10.1145/225014.225041.

[83] Stan Jarzabek. 2013. **Teaching Advanced Software Design in Team-Based Project Course**. In *International Conference on Software Engineering Education and Training* (CSEE&T '13), 31–40. DOI: 10.1109/CSEET.2013.6595234.

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

[85] C. W. Johnson and Ian Barnes. 2005. **Redesigning the Intermediate Course in Software Design**. In *Australasian Conference on Computing Education - Volume 42* (ACE '05). Australian Computer Society, Inc., 249–258. https://crpit.scem.westernsydney.edu.au/confpapers/CRPITV42Johnson.pdf.

[86] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. 2018. **Does Distance Still Matter? Revisiting Collaborative Distributed Software Design**. *IEEE Software*, 35, 6, 40–47. DOI: 10.1109/MS.2018.290100920.

[87] 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.

[88] 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.

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

[90] Daniel Kahneman. 2003. **Maps of Bounded Rationality: Psychology for Behavioral Economics**. *American Economic Review*, 93, 5, (December 2003), 1449–1475. DOI: 10.1257/000282803322655392.

[91] Sean H. K. Kang. 2016. **Spaced Repetition Promotes Efficient and Effective Learning: Policy Implications for Instruction**. *Policy Insights from the Behavioral and Brain Sciences*, 3, 1, 12–19. DOI: 10.1177/2372732215624708.

[92] 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.

[93] 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.

[94] 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.

[95] Chris F. Kemerer and Mark C. Paulk. 2009. **The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data**. *IEEE Transactions on Software Engineering* (TSE), 35, 4, 534–550. DOI: 10.1109/TSE.2009.27.

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

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

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. 2024. **Computer Science Curricula 2023**. Association for Computing Machinery. DOI: 10.1145/3664191.

[103] 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.

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

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

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

[107] Ze Shi Li, Nowshin Nawar Arony, Kezia Devathasan, and Daniela Damian. 2023. **"Software is the Easy Part of Software Engineering" — Lessons and Experiences from A Large-Scale, Multi-Team Capstone Course**. In *International Conference on Software Engineering: Software Engineering Education and Training* (ICSE-SEET '23). IEEE Press, 223–234. DOI: 10.1109/ICSE-SEET58685.2023.00027.

[108] Jenny T. Liang, Maryam Arab, Minhyuk Ko, Amy J. Ko, and Thomas D. LaToza. 2023. **A Qualitative Study on the Implementation Design Decisions of Developers**. In *International Conference on Software Engineering* (ICSE '23), 435–447. DOI: 10.1109/ICSE48619.2023.00047.

[109] José Lima, Paulo Costa, Thadeu Brito, and Luis Piardi. 2019. **Hardware-in-the-loop simulation approach for the Robot at Factory Lite competition proposal**. In *International Conference on Autonomous Robot Systems and Competitions (ICARSC 2019)*, 1–6. DOI: 10.1109/ICARSC.2019.8733649.

[110] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. 1996. Ariane 5 flight 501 failure report by the inquiry board. (1996). https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[111] Chris Loftus, Lynda Thomas, and Carol Zander. 2011. **Can Graduating Students Design: Revisited**. In *Technical Symposium on Computer Science Education* (SIGCSE '11). ACM, 105–110. DOI: 10.1145/1953163.1953199.

[112] 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.

[113] Matti Luukkainen, Arto Vihavainen, and Thomas Vikberg. 2012. **Three Years of Design-based Research to Reform a Software Engineering Curriculum**. In *Annual Conference on Information Technology Education* (SIGITE '12). ACM, 209–214. DOI: 10.1145/2380552.2380613.

[114] 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.

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

[116] Sanoop Mallissery and Yu-Sung Wu. 2023. **Demystify the Fuzzing Methods: A Comprehensive Survey**. *ACM Comput. Surv.*, 56, 3, Article 71, (October 2023), 38 pages. DOI: 10.1145/3623375.

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

[118] Tomi Mannisto, Juha Savolainen, and Varvana Myllarniemi. 2008. **Teaching Software Architecture Design**. In *Working Conference on Software Architecture* (WICSA '08), 117–124. DOI: 10.1109/WICSA.2008.34.

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

[120] 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.

[121] 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.

[122] 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.

[123] Christoph Matthies, Johannes Huegle, Tobias Dürschmid, and Ralf Teusner. 2019. **Attitudes, Beliefs, and Development Data Concerning Agile Software Development Practices**. In *International Conference on Software Engineering: Software Engineering Education and Training Track* (ICSE-SEET '19). IEEE Press, 158–169. DOI: 10.1109/ICSE-SEET.2019.00025.

[124] Christoph Matthies, Thomas Kowark, and Matthias Uflacker. 2016. **Teaching Agile the Agile Way — Employing Self-Organizing Teams in a University Software Engineering Course**. In *ASEE International Forum*. ASEE Conferences, (June 2016). DOI: 10.18260/1-2--27259.

[125] Mark E. McMurtrey, James P. Downey, Steven M. Zeltmann, and William H. Friedman. 2008. **Critical Skill Sets of Entry-Level IT Professionals: An Empirical Examination of Perceptions from Field Personnel**. *Journal of Information Technology Education: Research*, 7, 1, (January 2008), 101–120. DOI: 10.28945/181.

[126] Gerard Meszaros. 2007. **xUnit Test Patterns: Refactoring Test Code**. Addison-Wesley. http://xunitpatterns.com/Test%20Double.html.

[127] 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.

[128] Raffaella Negretti. 2012. **Metacognition in Student Academic Writing: A Longitudinal Study of Metacognitive Awareness and Its Relation to Task Perception, Self-Regulation, and Evaluation of Performance**. *Written Communication*, 29, 2, 142–179. DOI: 10.1177/0741088312438529.

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

[130] 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.

[131] 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.

[132] 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.

[133] Sofia Ouhbi and Nuno Pombo. 2020. **Software Engineering Education: Challenges and Perspectives**. In *Global Engineering Education Conference* (EDUCON '20), 202–209. DOI: 10.1109/EDUCON45650.2020.9125353.

[134] Wilson Libardo Pantoja Yépez, Julio Ariel Hurtado Alegría, Ajay Bandi, and Arvind W. Kiwelekar. 2023. **Training software architects suiting software industry needs: A literature review**. *Education and Information Technologies*. DOI: 10.1007/s10639-023-12149-x.

[135] 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.

[136] 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.

[137] Marian Petre. 2009. **Insights from Expert Software Design Practice**. In *Joint Meeting of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering* (ESEC/FSE '09). ACM, 233–242. DOI: 10.1145/1595696.1595731.

[138] Reinhold Plösch, Johannes Bräuer, Christian Körner, and Matthias Saft. 2016. **MUSE: A Framework for Measuring Object-Oriented Design Quality**. *Journal of Object Technology*, 15, 4, (August 2016), 2:1–29. DOI: 10.5381/jot.2016.15.4.a2.

[139] Carianne Pretorius, Maryam Razavian, Katrin Eling, and Fred Langerak. 2024. **When rationality meets intuition: A research agenda for software design decision-making**. *Journal of Software: Evolution and Process*, 36, 9, e2664. DOI: 10.1002/smr.2664.

[140] 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.

[141] Alex Radermacher and Gursimran Walia. 2013. **Gaps Between Industry Expectations and the Abilities of Graduates**. In *Technical Symposium on Computer Science Education* (SIGCSE '13). ACM, 525–530. DOI: 10.1145/2445196.2445351.

[142] 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.

[143] 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.

[144] 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.

[145] 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.

[146] 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.

[147] Çetin Semerci and Veli Batdi. 2015. **A Meta-Analysis of Constructivist Learning Approach on Learners' Academic Achievements, Retention and Attitudes**. *Journal of Education and Training Studies*, 3, 2, 171–180. DOI: 10.11114/jets.v3i2.644.

[148] Mary Shaw. 2000. **Software Engineering Education: A Roadmap**. In *Conference on The Future of Software Engineering* (ICSE '00). ACM, 371–380. DOI: 10.1145/336512.336592.

[149] Mary Shaw, Jim Herbsleb, and Ipek Ozkaya. 2005. **Deciding What to Design: Closing a Gap in Software Engineering Education**. In *International Conference on Software Engineering* (ICSE '05). ACM, 607–608. DOI: 10.1145/1062455.1062563.

[150] Mary Shaw and James E. Tomayko. 1991. **Models for undergraduate project courses in software engineering**. In *Software Engineering Education*. Springer Berlin Heidelberg, 33–71. DOI: 10.1007/BFb0024284.

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

[152] 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.

[153] 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.

[154] Antony Tang, Maryam Razavian, Barbara Paech, and Tom-Michael Hesse. 2017. **Human Aspects in Software Architecture Decision Making: A Literature Review**. In *International Conference on Software Architecture* (ICSA '17), 107–116. DOI: 10.1109/ICSA.2017.15.

[155] Antony Tang, Minh H. Tran, Jun Han, and Hans van Vliet. 2008. **Design Reasoning Improves Software Design Quality**. In *Quality of Software Architectures. Models and Architectures* (QoSA '08). Springer, 28–42. DOI: 10.1007/978-3-540-87879-7_2.

[156] Saara Tenhunen, Tomi Männistö, Matti Luukkainen, and Petri Ihantola. 2023. **A systematic literature review of capstone courses in software engineering**. *Information and Software Technology*, 159, 107191. DOI: 10.1016/j.infsof.2023.107191.

[157] Charles Thevathayan and Margaret Hamilton. 2017. **Imparting Software Engineering Design Skills**. In *Australasian Computing Education Conference* (ACE '17). ACM, 95–102. DOI: 10.1145/3013499.3013511.

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

[159] Christopher S. Timperley, Gijs van der Hoorn, André Santos, Harshavardhan Deshpande, and Andrzej Wąsowski. 2024. **Robust: 221 bugs in the robot operating system**. *Empirical Software Engineering*, 29, 3, 57. DOI: 10.1007/s10664-024-10440-0.

[160] Dan Tofan, Matthias Galster, and Paris Avgeriou. 2013. **Difficulty of architectural decisions – a survey with professional architects**. In *Software Architecture*. Springer, 192–199.

[161] Hans van Vliet and Antony Tang. 2016. **Decision making in software architecture**. *Journal of Systems and Software*, 117, 638–644. DOI: 10.1016/j.jss.2016.01.017.

[162] Vasudeva Varma and Kirti Garg. 2005. **Case Studies: The Potential Teaching Instruments for Software Engineering Education**. In *International Conference on Quality Software* (QSIC '05), 279–284. DOI: 10.1109/QSIC.2005.18.

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

[164] 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.

[165] Ian Warren. 2005. **Teaching Patterns and Software Design**. In *Australasian Conference on Computing Education - Volume 42* (ACE '05). Australian Computer Society, Inc., 39–49. https://crpit.scem.westernsydney.edu.au/confpapers/CRPITV42Warren.pdf.

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

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

[168] 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.

[169] Bian Wu and Alf Inge Wang. 2012. **A Guideline for Game Development-Based Learning: A Literature Review**. *International Journal of Computer Games Technology*, 2012, 1, 103710. DOI: 10.1155/2012/103710.

[170] Stephen S. Yau and Jeffery J.-P. Tsai. 1986. **A Survey of Software Design Techniques**. *Transactions on Software Engineering* (TSE), SE-12, 6, 713–721. DOI: 10.1109/TSE.1986.6312969.

[171] Jianmin Zhang and Jian Li. 2010. **Teaching Software Engineering Using Case Study**. In *International Conference on Biomedical Engineering and Computer Science* (ICBECS '10), 1–4. DOI: 10.1109/ICBECS.2010.5462378.

[172] Li Zhang, Yanxu Li, and Ning Ge. 2020. **Exploration on Theoretical and Practical Projects of Software Architecture Course**. In *International Conference on Computer Science & Education* (ICCSE), 391–395. DOI: 10.1109/ICCSE49874.2020.9201748.

[173] Xuemei Zhang and Hoang Pham. 2000. **An analysis of factors affecting software reliability**. *Journal of Systems and Software*, 50, 1, 43–56. DOI: 10.1016/S0164-1212(99)00075-8.

[174] Celal Ziftci and Ben Greenberg. 2023. **Improving Design Reviews at Google**. In *International Conference on Automated Software Engineering* (ASE '23), 1849–1854. DOI: 10.1109/ASE56229.2023.00066.