Summer 7-1-2014

INVARIANT INFERRING AND MONITORING IN ROBOTIC SYSTEMS

Hengle Jiang
University of Nebraska-Lincoln, jianghengle@gmail.com

INVARIANT INFERRING AND MONITORING IN ROBOTIC SYSTEMS

by

Hengle Jiang

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Sebastian Elbaum

Lincoln, Nebraska

May, 2014

INVARIANT INFERRING AND MONITORING IN ROBOTIC SYSTEMS

Hengle Jiang, M.S.

University of Nebraska, 2014

Adviser: Sebastian Elbaum

System monitoring can help to detect abnormalities and avoid failures. Crafting monitors for today's robotic systems, however, can be very difficult due to the systems' inherent complexity and its rich operating environment.

In this work we address this challenge through an approach that automatically infers system invariants and synthesizes those invariants into monitors. This approach is inspired by existing software engineering approaches for automated invariant inference, and it is novel in that it derives invariants by observing the messages passed between system nodes and the invariants types are tailored to match the spatial, time, temporal, and architectural attributes of robotic systems. Further, our approach automatically classifies and synthesizes invariants into a monitor node that can be seamlessly integrated into systems built on top of publish-subscribe architectures. The monitor can be also tailored to trigger actions when an invariant is violated. We have assessed the approach in the context of three UAV systems to better understand its potential. In our case study, we found that invariants can be useful for developers and that the synthesized monitor can reduce system failure rate when facing unexpected faults from 76.2% to 10.6%.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr. Sebastian Elbaum who showed me the road and helped to get me started on the path to this degree. His patience, enthusiasm, encouragement and faith in me throughout have been extremely helpful. He was always available for my questions and he was positive and gave generously for his time and vast knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my M.S. study.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Carrick Detweiler and Dr. Matthew Dywer, for their encouragement, insightful comments, and hard questions.

My sincere thanks also goes to Dr. Carrick Detweiler, for offering me the opportunities working in Nimbus Lab and leading me in diverse exciting projects.

Thanks to John-Paul Ore for sharing his water sampling project data, and to David Anthony for sharing his crop surveying project data to use in my thesis.

I thank my fellow labmates: Andew Mittleider, Megan Jensen, Adam Taylor, Jinfu Leng, Sreeja Bannerjee, Charlie Lucas, Javier Darsie and Brent Griffin, for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years. In particular, I am grateful to Dr. Ying Lu for leading me into the computer science world.

Last but not the least, I would like to thank my wife Liqun Bi for supporting me spiritually always throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Monitoring a system for anomalies is a common approach to detect conditions that may lead to failures and to take corrective actions. Such monitors must be carefully crafted by engineers with the domain knowledge to understand what could constitute abnormal behavior. This process becomes increasingly challenging as the monitored system and its operating environment grow in complexity.

Consider, for example, the scenario illustrated in Figure 1.1 where a small Unmanned Aerial Vehicle (UAV) is autonomously following and attempting to land on a moving platform whose location is continuously fed to the UAV. A typical landing test consists of placing the UAV starting a few meters away from the platform, finding and following the moving platform, and then initiating the landing sequence. Using a message passing system middleware such as ROS (Robot Operating System)[12], an implementation of this system contains several distributed processes that communicate through dozens of message channels.

An engineer developing a monitor to detect anomalies for this kind of system is likely to focus on a small subset of variables and relationships between variables. For example, a monitor crafted for this system would likely check whether the

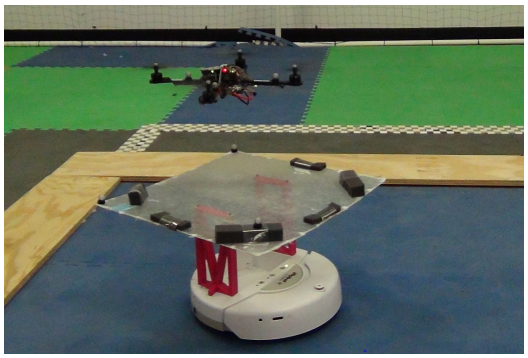

Figure 1.1: UAV attempting to land on moving platform.

positions of the UAV and the platform are aligned when landing is initiated, and whether the speed of the platform is less than a safe maximum. There are, however, many other aspects of the system worth monitoring that are more subtle and may not be considered by the engineer given the number of variables and relationships involved. For example, it may help to ensure that the platform is horizontal and not rotating when landing, the UAV's angles are not greater than a multiple of the UAV's commanded velocity, there is only one landing platform reporting its location, and the platform is unoccupied and able to support the weight of the UAV. In addition, the UAV operating in different scenarios may carry varied safe conditions. For instance, the safe angle of the UAV flying in a no-wind environment is quite different from it in a strong-wind environment. Similarly, when the platform is moving on different types of terrain, the UAV's safe landing conditions are quite different.

As the system complexity increases, it is unlikely that the engineer will consider all possible variables or relationships in all scenarios. To alleviate this challenge, we propose an approach to automate the synthesis of monitors from the traces gathered by this type of distributed system operating a robot.

Our approach is inspired by existing software engineering approaches for

automated invariant inference[18]. The core idea of this type of approach is to infer system invariants from traces collected during system execution, iteratively instantiating potential invariants from a set of invariant templates utilizing the trace values, and dropping or refining the ones that are falsified by other trace values.

For example, given a template invariant $varX \geq constant$ and a trace of six variable-value pairs collected from time $t1$ to time $t6$, $tr = \{t1 : a = 1, t2 : b = 3, t3 : a = 1, t4 : a = 2, t5 : a = 1, t6 : a = -1\}$, the approach would instantiate the invariant template as $a \geq 1$ after reading the value of a at $t1$ and further support it until $t6$ when value $a = -1$ is observed. Then it becomes necessary to refine the invariant to $a \geq -1$. For variable b an invariant may not be reported as there may not be enough values to support that instantiation. Given a set of traces, the inferred invariants provide a characterization of the behavior of the system as exhibited in those traces, and can be the basis for determining what to monitor and what is an anomaly.

Existing techniques to automatically infer invariants have been shown useful for generating generic invariants like the one illustrated above to act primarily as a function's pre and post conditions. The application of these techniques to large distributed robotic systems, however, has been limited. We conjecture that this is due to the focus on the generation of low level invariants which is impractical for these large systems, the lack of domain-specific invariants that capture the temporal and spatial aspects of robotic systems, and the lack of tools to seamlessly integrate such approaches into the development process and common toolsets. In this work we set out to tackle these challenges.

1.1 Approach Overview

The goal of our approach is to enable the automatic generation of system monitors that can detect anomalous behavior and launch counter-measures. The type of system we target is a robotic system made of distributed nodes that sense, actuate, and communicate through some form of message passing scheme. Our work was motivated and implemented in the context of ROS [12], but the approach is generalizable to other similar message passing infrastructures, as well as service oriented architectures. (e.g., LCM [5], Microsoft Robotics Developer Studio [6] , CLARAty [2]). Note that we operate at the granularity of messages commonly used by robotic systems operating under a publish and subscribe architecture. This reduces the monitoring overhead and it lets us infer properties related not just to program states, but also to message sequences, which are critical to robotic systems.

Figure 1.2 provides an overview of the approach, which is conceptually similar to what is currently performed by existing dynamic invariant inference frameworks [18, 19, 20, 28]; we have highlighted the differences by bolding certain components' labels.

As shown in Figure 1.2, system S, a configuration file(CFG) and a training set TS, serve as the only inputs to the approach, and the whole workflow can be separated into two parts: Invariant Inference and Monitor Synthesis. In the first part, S is instrumented to capture the messages passed between the nodes in the system, constituting system S'. When S' is executed with the training set TS, a set of $|TS|$ traces($Traces$) is generated, where each trace will contain a sequence of variable-value pairs found in the messages. The approach will then attempt to instantiate the predefined invariant templates based on the information

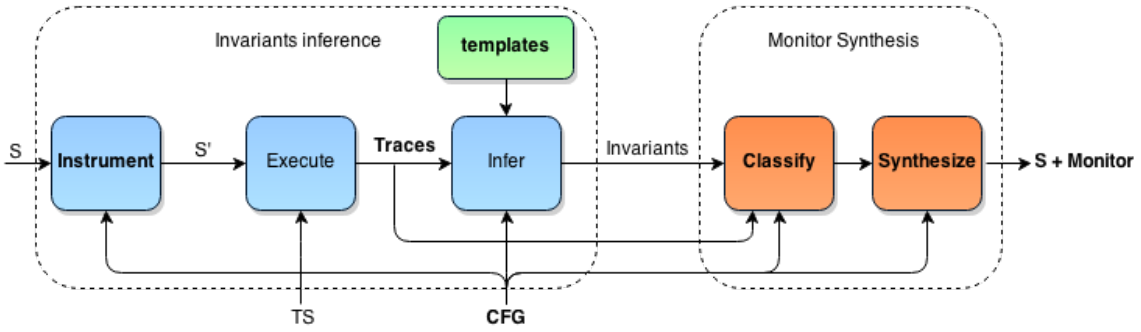

Figure 1.2: The whole workflow

found in *Traces* and *CFG*. Each instantiated invariant is a boolean expression that characterizes the variables values observed in *Traces*. In the second part, the invariants are further pruned with data traces based on their classifiers' capabilities. Then, the invariants are synthesized into a monitor that can be incorporated into the system *S*.

1.2 Thesis Structure

Through this work we aim to make automated invariant inference techniques amenable to robotic systems. Our main contributions are:

- With just message passing channels instrumentation, our approach captures critical information in robotic systems, and organizes them into data traces in a way to enable more interesting invariant inference.
- We have developed invariant templates that account for properties that are deemed important in the context of robotic systems, such as those characterizing the relationship between variables that have a continuous distribution such as sensors values, those including a time component to capture the derivatives of raw variable values, those that can differentiate among system

operating modes, those characterizing the architecture of the robotic system, and those capturing temporal properties of the program behaviors.

- We have implemented a version of the approach that automatically classifies and synthesizes invariants into a monitor node that can be seamlessly integrated into existing ROS systems. The monitor can be tailored to trigger actions when an invariant is violated.

- We have assessed the approach in the context of three UAV systems to better understand its potential. In our experiments, we found that the monitor can reduce system failure rate when facing unexpected scenarios from 76.2% to 10.6%. In addition, it also can be used to check user expectations and assumptions. The new developed invariant templates have potentials to detect subtle problems in Robotic Systems.

The thesis is organized into the following chapters. In Chapter 2 we will introduce the background and the related work. In Chapter 3 we will describe how we perform Invariant Inference, including program instrumentation, trace generation, and invariants templates. In Chapter 4, we will discuss our Monitor Synthesis, including invariant classification and monitor generation. In Chapter 5, we present three case studies to evaluate our approach. In Chapter 6, we conclude and discuss future work.

Chapter 2

Background and Related Work

Our work aims to enable the automatic generation of system invariant monitors that can detect anomalous behavior in distributed robotic operating systems. Since we implemented our tool specifically on ROS (Robotic Operating System), we first introduce the background of this robotic system, and then explore the related work in three contexts: invariant detection and monitoring, robot execution monitoring, and distributed system debugging. A summary of the related work is described in Table 2.1.

2.1 ROS

ROS[12] is a software framework for robot software development, which provides operating system-like functionality on a heterogeneous computer cluster. It is based on a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor, control, state, planning, actuator and other messages.

Figure 2.1 shows a publish/subscribe graph view of a ROS program, where

Context	Other Approaches		Our Approach
Invariant Inference and monitoring	Daikon[18, 19]	Powerful invariant inference engine and framework; sophisticated toolset with multiple front ends and extension capabilities.	Built on Daikon's framework; richer invariant templates; Daikon's front end for ROS.
	DIDUCE[28]	Online invariant inference and monitoring on Java bytecode; simple invariant templates on one variable or expression.	Offline invariant inference and runtime monitoring on robotic systems; richer invariant templates.
	PRECIS[34]	Invariant inference only; clusters variable's values by path information in terms of predicate words; needs to access source code.	Invariant monitoring also; simply clusters data by enumerable variables; does not need to access source code.
	DySy[15]	Invariant inference only; uses simultaneous symbolic execution to get more abstract and general invariants; needs to access source code.	Invariant monitoring also; no static analysis; does not need to access source code.
	Javert[20]	Mines small generic temporal patterns from event sequences and composes them to construct large, complex patterns.	Infers temporal invariants like ordered-pair intervals, which is a simple but effective temporal property for robotic systems.
	GK-tail[16]	Generates extended finite state machines (EFSMs, annotating FSM edges with transition conditions on data values) from interaction traces.	Only detects the existence of event patterns in the form of regular expressions defined by users.
Robotic Execution Monitoring	GSOLR[23, 24]	Model-based analytical approach; uses reachability analysis to guarantee safety against worst-case disturbances.	Data-driven approach; learns system properties and enforces these properties.
	Self-Awareness Model[25, 26]	Data-driven approach; detects faults based on the inherent dynamics of inter-component communication.	Data-driven approach; detects faults based on both temporal and state properties of inter-component communication.
	Real-time Diagnosis & Repair[22]	Uses model-based diagnosis for fault detection and localization, and a repair module executes an appropriate action to recover the system from the fault.	Localizes faults on topics or services; monitor can take isolation or other recovery action.
Distributed System Debugging	Pip[35]	Accepts expectations in a declarative language from users; logs actual system behaviors and explores expected and unexpected behaviors.	Infers and enforces structure and performance specifications.

Table 2.1: Comparison of our approach and others

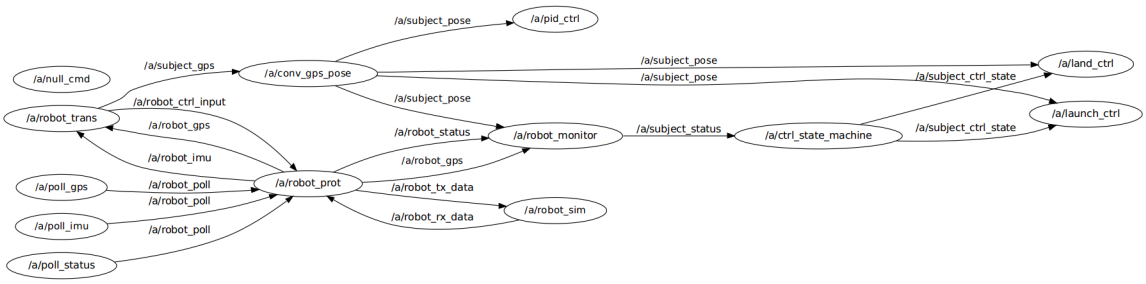

Figure 2.1: Graph view of a ROS program

ellipses represent nodes and directed edges are communication channels (publish/subscribe) between nodes. Nodes are the basic elements in a ROS program, and each node is simply a process launched in a ROS system. Nodes can communicate with each other through two mechanisms: topic and service. A topic works like a message bus, where nodes can publish to and subscribe from, while the messages published on topics are just user defined data structures. As shown in Figure 2.1 the label on the edge is the topic name, and the edge starts from the publisher who will publish message to the topic, and ends to the subscriber who will subscribe message from the topic. Note that multiple nodes can publish or subscribe to one topic, so that topic may build multiple communications. On the contrary, a service can only be provided by one server node at a time, and the client nodes can communicate with the server node by calling the service. All these communications are registered and directed by a ROS master node.

ROS also provides several ways to configure the program at launch time and runtime. The sample XML-format launch file in Figure 2.2 can configure and deploy multiple nodes with particular arguments and parameters at launch time. The launch file makes the ROS program more flexible since it enables launching multiple nodes from one package, remap resource names (node, topic, service and parameter) and set parameters. For example, in Figure 2.2 six nodes would

```
<launch>
  <group ns="a">
    <node pkg="vicon_pose" type="vicon_pose" name="conv_gps_pose">
      <remap from="subject_pose" to="car_pose"/>
      <param name="object_name" value="car" type="str" />
    </node>
    <node name="robot_prot" type="arb_subject_ctrl" pkg="arbitration">
      <remap from="shape_shifter" to="quad_ctrl_input"/>
      <rosparam param="state_mappings">
        [[ "null_input", [ 0, 1, 4 ] ],
         [ "ignition_input", [ 2, 3 ] ],]
      </rosparam>
    </node>
    <node name="robot_monitor" type="asctec_monitor" pkg="supervision"/>
    <node name="ctrl_state_machine" type="manage_subject_ctrl" pkg="state_machine"/>
    <node name="poll_gps" type="rostopic" pkg="rostopic" args="pub -r 20.0
      'robot_poll' collab_msgs/QuadPoll '{header: auto, pitch: 0.0,
       roll: 0.0, yaw: 1.0, thrust: -1.0}'"/>
    <node name="server" type="add_two_ints_server.py" pkg="test_srv"/>
    <include file="$(find collab_launch)/common.launch" />
  </group>
</launch>
```

Figure 2.2: Launch file of a ROS program

be launched together explicitly. The first node *conv_gps_pose* launched from an executable file *vicon_pose* in *vicon_pose* package. Inside the scope of this node, the resources it refers as *subject_name* would be remapped to a new name *car_pose*. And it also sets a string parameter *object* with the value of "*car*". In addition, the ROS parameter server also provides a way to access and change parameters at runtime.

2.2 Invariant Inference and monitoring

Our work was inspired in part by the evolution and maturity gained by techniques and tools available to infer likely program invariants. Our technique and tool build specifically on Daikon[18, 19], one of the pioneer approaches with probably the most sophisticated toolset openly available. The likely invariants produced by Daikon are a dynamically-generated analogue of a program specification, which is valuable in software testing, debugging and verification. We discuss Daikon in more detail in Section 3.1.

Among invariant monitoring, DIDUCE[28] was the first work to use invariants for runtime monitoring and diagnosis, which is similar to our work. DIDUCE

needs to instrument java bytecode, focuses on program states at particular program points (procedure calls and heap accesses), and relaxes invariants or reports to users when detecting anomalies. Instead, our approach, operating in the context of message-passing and service-oriented architectures supporting distributed robotic systems, does not instrument the program source code, but just focuses on messages to detect system anomalies by richer types of invariants, and takes corrective measures (like interrupting the message passing) to prevent system crashes.

Still, several other complementary efforts have emerged in the last few years, ranging from refining state invariants to temporal and behavioral model inferences[20, 40, 15, 34]. To improve invariant generation, researchers have taken advantage of static analysis to guide the dynamic invariant inference. For example, PRECIS[34] proposed generating invariants through program path guided clustering. Their approach records inputs and outputs together with predicates for branch conditions, and uses linear regression on inputs and outputs grouped by predicate words to infer path invariants. DySy[15] uses symbolic execution to infer more general invariants, as it combines concrete executions of actual test cases with simultaneous symbolic executions of the same tests to produces abstract conditions as program invariants. In our work, we did not apply static or symbolic analysis, because we are facing large scale distributed robotic systems instead of a class or a function. Consequently, our approach is a purely based on traces without any dependence on source code. However, to extend the power of Daikon, we also use a cluster analysis on the trace data to generate conditional invariants (see Section 3.6).

Temporal invariants represent some rules on events' order. Javert[20] is one tool that can extract and compose temporal patterns from event traces, and its extension allows for simultaneously learning and enforcing general temporal properties over

method call sequences[21]. We have also implemented such temporal invariant inference on publishing messages and calling services. We specifically infer the ordered-pair interval invariant, which tells that an event always happens after another event within certain time and events happening in the interval.

In terms of behavioral model inference, researchers have focused on interactions between components, and the results are usually in the form of finite state machines (FSM). Lorenzoli et al. have developed a dynamic analysis algorithm called GK-tail combining the ideas of invariant detection and temporal property mining[16]. The result of this kind of inference is a extended finite state machines (EFSMs). In our approach, we can also detect existence of event patterns in the form of regular expressions defined by users. However, we did not yet explore FSM or EFSMs inference in this work.

2.3 Robot Execution Monitoring

In the context of robotic systems, monitoring for error detection is a well known area [38]. The potential for missing information, unreliable and imprecise sensors, and the stochastic nature of the operating environment often makes monitors a necessity. Existing efforts can be categorized into model-based or data-driven, based on how to build the system model (invariants in our approach) to detect anomalies.

Model-based approaches follow either an analytical or knowledge-based method, where developers model each state beforehand and use this model to estimate the current system state (i.e., normal or faulty). Analytical approaches are commonly used in the design of control systems, where the models are constructed based on fundamental assumptions. They are precise and mostly targeted at problems fairly

close to the hardware as well as to the raw sensor data.

In the context of quad rotors similar to the ones we used in our study, there have been several recent efforts that attempt to detect anomalies by model-based approaches. For example, Gillula and Tomlin[23] proposed a framework using reachability analysis in a way that prevents the control system from taking an unsafe action. They further proposed an adapted form of their approach called GSOLR by modeling the worst-case disturbance state-dependent manner learned online[24].

The data-driven approach does not need a model beforehand; instead it tries to infer an abstract model (usually a statistical model) of the original system from the data, and uses the inferred model to detect faults. Golombek and Wrede et al. presented a so-called self-awareness model [25, 26]. It also requires a message-passing robotic systems, and it maps each system's internal data exchange to an event (such as *Component*1 *updates sensorA on Component*2). And then it infers a probabilistic model on the event sequences, which is the histogram of the interval time between any two events, by which it could detect many errors, such as component failure, resource starvation and asynchronous communication. Our approach also infers such temporal invariants as the ordered-paired interval invariant (see Section 3.4.4.1), which captures not only the time but also the events happening in the interval. Besides temporal properties, our approach also captures a variety of state properties.

After an anomaly is detected, existing efforts have been designed to perform diagnosis and remediation based on models defined by domain experts. Steinbauer and Morth presented a solution for real-time fault detection and repair of control softwares of autonomous robots[22]. Their diagnosis system uses model-based diagnosis for fault detection and localization, and a repair module executes an

appropriate action to recover the system from the fault.

Our approach is complementary to these approaches, and unique in that it can generate more general invariants that were not considered by domain experts, not defined by simple statistics, and that may be relevant to many robotic systems as they are instantiated by a training set. Furthermore, the implementation within ROS makes it directly applicable to a large set of existing robotic systems.

2.4 Distributed System Debugging

In distributed system debugging, developers usually focus on two kinds of bugs: structure bug and performance bug[35]. A structural bug results in processing or communication happening at the wrong place or in the wrong order. Most of these approaches[35, 36, 32] collect event sequences as causal paths, and check expectations or anomalies as errors. Inspired by their approaches, we build our structure invariants as architecture invariants (see Section 3.4.3), which indicate the correct communication model of the ROS programs. However, our approach does not collect causal paths driven by events, but simply records the publish/subscribe architectures and infers the invariants. Compared to their approach, ours relies less on the program instrumentation and the behavior expectation.

The performance bugs result in processing consumes too much or too little of some important resources, for example time. The measure of the performance also depends on the causal paths collected [35, 31, 36, 32]. Our approach also records and infers some invariants of system performance. For example, we record ROS service calls and infer invariants on the response delay (see Section 3.4.1). Another performance measure is message latency(see Section 3.4.1), which constrains the delay between publish and subscribe.

Chapter 3

Invariant Inference

In this chapter, we will describe the invariant inference process in detail, which is the core part of our approach. As shown in Figure 1.2, the input is a system, a training set and a configuration file, and the output is a set of invariants. The invariant inference process can be divided into three steps: the first step enables the generation of data traces, the second is translating the data traces to feed the extended invariant inference engine Daikon, and the third is the actual invariant inference. The goal of this process is generating as many as possible meaningful and useful invariants, while suppressing trivial or redundant invariants. To achieve this goal, we enrich the sources of information of the data traces in the first step. Then, we organize the data trace in a way that Daikon can capture more interesting invariants. For the last step, we develop new invariant templates to increase the power of the inference process. In the following sections, we will illustrate these steps in more detail. We start with describing the invariant inference engine Daikon, on which we base our work.

3.1 Extending Daikon

Daikon is an implementation of dynamic detection of likely invariants. It provides a flexible framework with a sophisticated toolset for invariant inference. The framework of Daikon is shown in Figure 3.1. Daikon provides several language-specific front-end tools for program instrumentation and an extensible invariant template set. A front-end tool puts probes in the target program (more specifically at methods' entries and exits), and creates a *.decl* [3] file containing the declarations of the program points and the variables associated with them. During execution, these probes output variables' values on their program points to the data traces in Daikon's input format (*.dtrace* files [3]). At inference time, the invariant templates are used to initialize and check the invariants on the data trace, and developers can extend this template set. Generally, the inference engine follows these steps:

- Given variables declared in the *.decl* file and Daikon's parameters in the settings file, it initializes all possible invariants on the variables based on the invariant templates;
- It reads the data from the *.dtrace* file, checks all the invariants initialized in the first step, and dismisses or refines them if the data in the trace violates them;
- After finishing reading all the trace data, it filters out unjustified or redundant invariants based on the settings, and finally outputs all the invariants remaining.

Our work builds specifically on Daikon in the context of ROS. From the perspective of Daikon's framework, we have built a ROS front end, as we have tailored Daikon for ROS and develop new invariant templates.

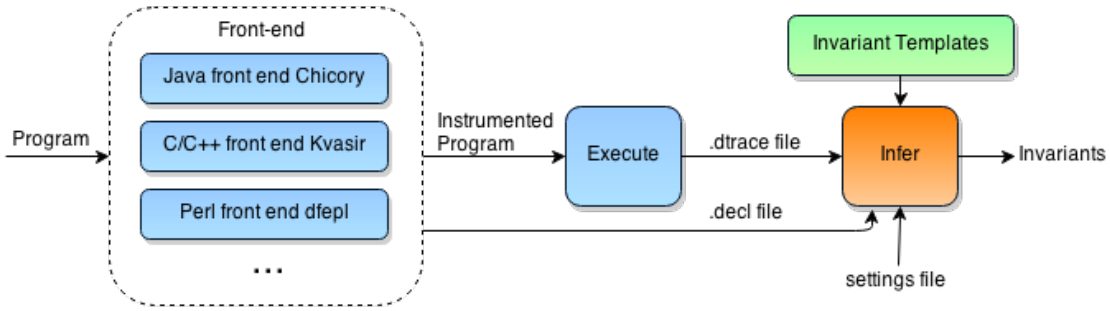

Figure 3.1: Daikon Framework

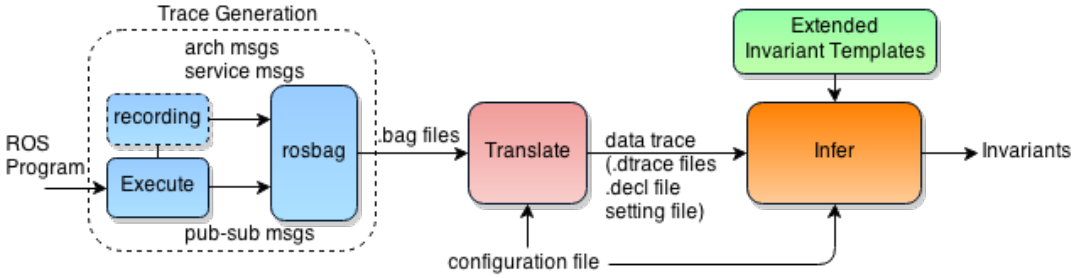

Figure 3.2: Invariant Inference Work Flow

Figure 3.2 shows the inference workflow of our approach which extends Daikon's framework. To get the invariants, our approach needs to go through three steps: trace generation, trace translation, and invariant inference. First, the target program's executions are recorded into traces, then our approach translates these traces into Daikon's trace format, and finally Daikon uses our extended invariant templates to infer and output the invariants.

The inference process can be configured by a XML-format configuration file as shown in Figure 3.3, in which the *scope* tag defines how the messages will be extracted and organized, and the *detect* tag tells where to find the original data files. Our approach works at the granularity of messages, so the *scope* tag defines the sources (like topics in ROS system) we want to extract messages from and how to organize these messages into data traces. The *scope* tag also declares the inference

```
−<imROS project="demo">
 −<scope>
  −<publish id="a" relative="1">
     <topic> /a/cmd_subject_ctrl_state </topic>
   </publish>
  −<publish id="b" conds="1" relative="1">
     <topic> /a/task_waypose </topic>
   </publish>
   <call id="c"> /a/execute_task </call>
   <arch id="d"> pub,sub,srv </arch>
  −<temporal id="e" events="a,b,c">
     <pattern> (AB)* </pattern>
   </temporal>
  −<condition id="1">
     <topic> /a/cmd_subject_ctrl_state </topic>
     <value> state == 4 </value>
   </condition>
 </scope>
−<detect inv="demo.inv">
   <bag> bags/demo </bag>
 </detect>
</imROS>
```

Figure 3.3: Inference Part of a Configuration File

of other two kinds of invariants: architecture invariants and temporal invariants, which will be described in the following sections. The *detect* tag indicates the training data set (specifically the bags files of the ROS system) for the invariant inference. We will explain these in detail shortly.

We contribute three key extensions. First, we perform data capture at the level of the structured messages that are sent between ROS nodes. We observed that these higher level messages cause less overhead while still providing a rich enough data set from which to generate invariants on a per-topic level. Another advantage of this shift is that we can use a common ROS tool called rosbag to record most data without instrumenting the program, which means we can avoid instrumenting and adding overhead to complex source code. For some particular invariants (service call and architecture) we still need to add a node to retrieve the desired information, but we can do it through the modification of the program's launch file without instrumenting the source code. Second, we group messages according to their topics in the publish/subscribe graph, which helps to capture interesting properties across multiple topics. Third, we extend the invariant template set with

four new invariants: time-related, polygon, architecture, and temporal, which are particularly useful in Robotic Systems. Plus, we extend the invariant justification procedure of our new polygon invariants, which can help when the polygon invariants grow explosively. We also implement a simple cluster method to infer more precise invariants on separate sub traces.

3.2 Trace Generation

As mentioned before, we perform data capture at the level of the structured messages that are sent between the ROS nodes. While avoiding source code instrumentation, it provides a rich enough data set for invariant inference. The goal of trace generation is collecting all relevant data into traces. In addition to the normal messages communicated between nodes, our approach enriches the traces by collecting service messages, ROS parameters, and ROS architecture. All that special information also goes into messages to be recorded.

We want each message in the generated traces to include the time stamp, the message type, the message value, and the source of the message (topic). ROS's rosbag tool meets our requirements to capture the messages published through topics, but it misses information from other sources such as service messages, ROS parameters, and ROS architectures, so we need to extend message traces with an additional node that we call the recording node.

Services provide another way for nodes to communicate with each other. However, capturing service invocations is challenging, because a ROS service works like a point-to-point private communication between nodes. In order to record service usages, our approach has to intercept the service connection. It first needs to query the ROS master node to get all the services. Second, for each

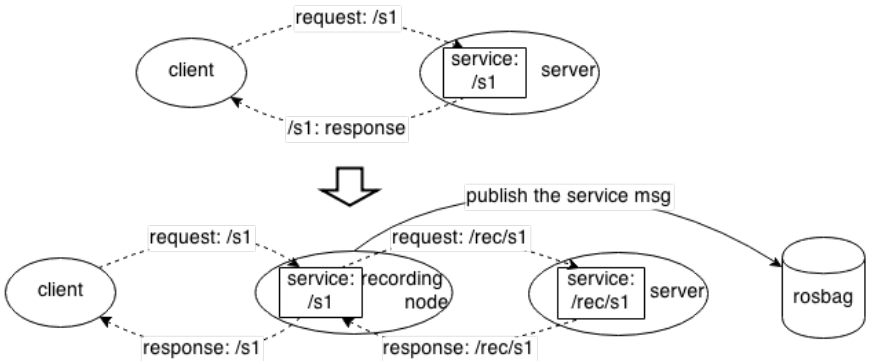

Figure 3.4: Example of remapping service

service, our approach remaps it to a new service name for all its server nodes. Third, the recording node provides the original services by relaying the request and response messages between the client and the real server. As shown in Figure 3.4, in the original system, the client node calls a service named $/s1$ on the server node. Our approach remaps the service with another name $/rec/s1$ for the server node, and makes the recording node relay the service. In this way, every time a service is called, the recording node publishes the service messages including the request and response messages, the client node, the real server node and the response time, and the rosbag tool records the service message into the trace. This service relaying introduces addition response delay.

ROS parameters work like system environment variables for ROS systems. For the system in Figure 5.15, we can use global parameters to declare the initial GPS coordinates, and the nodes' private parameters can configure the nodes behavior like message publishing rate. To collect them, as shown in Figure 3.5 our recording node queries the master to get all the parameters which have been set on the parameter server, and then publishes them to the special parameter topic. It performs the query at a configurable time interval, and if any parameter has been changed, it republishes the parameter with the updated value. In this way, our

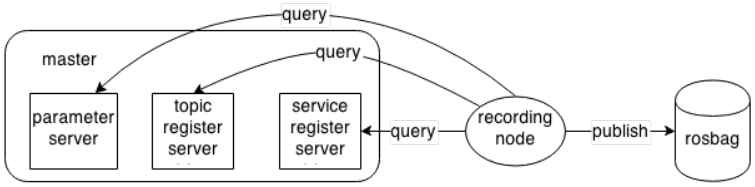

Figure 3.5: Architecture and Parameter recording

approach records them into the trace through the rosbag tool.

The system architecture is important information to collect, which indicates the nodes' publish and subscribe structure represented by a graph of nodes and topics or services, as will be addressed shortly in Section 3.4.3. The architectural information is recorded by the recording node as shown in Figure 3.5, which queries the ROS master to get the current architecture (topic and service), and then publishes the information to the architecture topic, which is recorded by the rosbag tool. The recording node takes a snapshot of the architecture at a configurable time interval, and if this snapshot is different from the previous one, it publishes the new architecture. The topic architecture information is presented as a map of topics to publishers and subscribers: $\{topic1 : \{pubs : \{pub1, pub2, \ldots\}, subs : \{sub1, sub2, \ldots\}\}, \ldots\}$; while the service architecture information is just a map of services to servers: $\{service1 : server1, service2 : server2, \ldots\}$.

3.3 Trace Translation

The goal of the trace translation step is translating the messages in bag files into Daikon's format. Moreover, it organizes the data to let Daikon infer more interesting invariants.

Variables in Daikon's format must be grouped by locality also known as program points. For example, method entry and exit points are considered

program points by Daikon's inference engine. Only variables at the same program point are analyzed together to compute invariants. For example, Daikon would infer the relationship between the variables $x1$ and $x2$ on the entry point of method A, but it would never infer any relationship between $x1$ or $x2$ at entry point of method A and the variable y at entry point of method B. In our case, we do not explicitly have program points in terms of methods' entries or exits; however, we have processing nodes and topics. Our approach clusters topic messages consumed and published by a node to identify invariants for the node. The idea is that the entry values in the messages consumed by a node are likely to define its behavior and affect its outputs as evident in the published messages.

For example, in the ROS system shown in Figure 3.6, each ellipse represents a node, rectangles show topics, and the directed edges tell the publish and subscribe relations. We can see that the topic */a/cmd_subject_ctrl_state*⓪ is published by the node */a/car_ctrl*① which subscribes to the topics */a/car_pose*①, */a/subject_ctrl_state*① and */a/subject_pose*①. We group the messages on these four topics together at the program point of */a/cmd_subjec_ctrl_state*, because the messages on the topic */a/cmd_subject_ctrl_state* are probably dependent on these on the three input topics. Since dependencies may exist across more than one publisher, we can further group topics along the chain. For instance, if we cluster messages across two publishers, the */a/cmd_subjec_ctrl_state* program point will contain six topics, where the two extra topics are */a/subject_status*② and */vicon/car*②. Although the long-chain grouping may present interesting invariants, it also introduces increasing overhead as we will infer invariants on more variables. And the relations between topics across more publishers are likely to be weak.

The process of pairing messages for this is illustrated in Figure 3.7. The

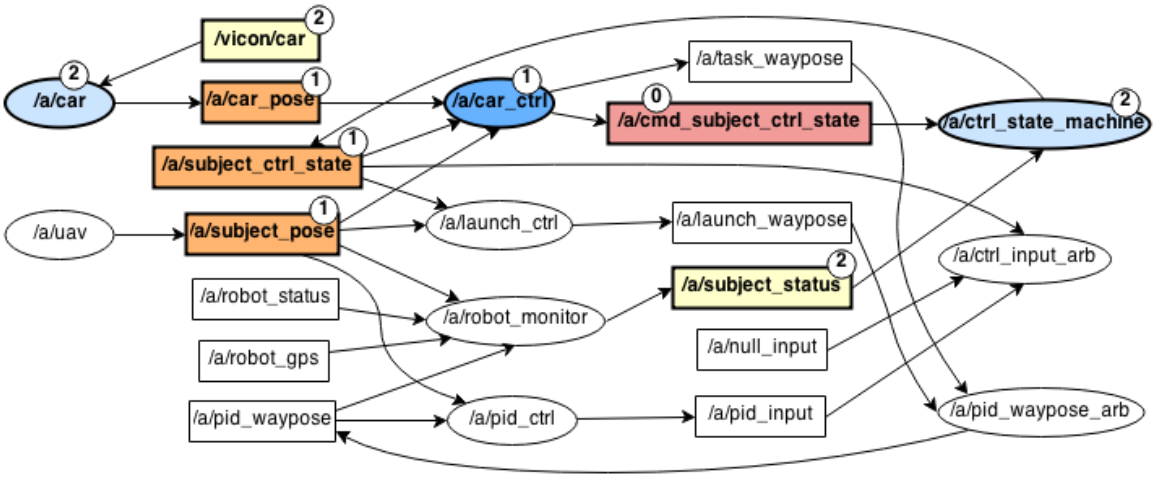

Figure 3.6: A ROS Program Example

top part presents a partial message trace, and the bottom part shows the data trace of program points of *publish/a/cmd_subjec_ctrl_state*. In each pairing, a published message on topic */a/cmd_subject_ctrl_state* is paired with the latest values of messages on several topics including */a/subject_ctrl_state*, */a/car_pose* and */a/subject_pose*. Note that not all messages are published at the same rate, so each pairing includes the published message with the latest value available for all the incoming messages. Our approach can be parameterized to relate published values to a range of previously consumed values.

As shown in Figure 3.7, our approach attaches a time stamp to every program point. Extracted from the bag file, the time stamp indicates when the message is published, and it can be used to infer time-related invariants (Section 3.4.1).

For services, the translator retrieves the messages (request and response) on the service calls from the service-recorded topic, and puts these messages into corresponding service program points as shown in Figure 3.8. ROS parameters are much like global variables which can be put into any program point if the user chooses. For architectural messages, our approach reads and converts them into

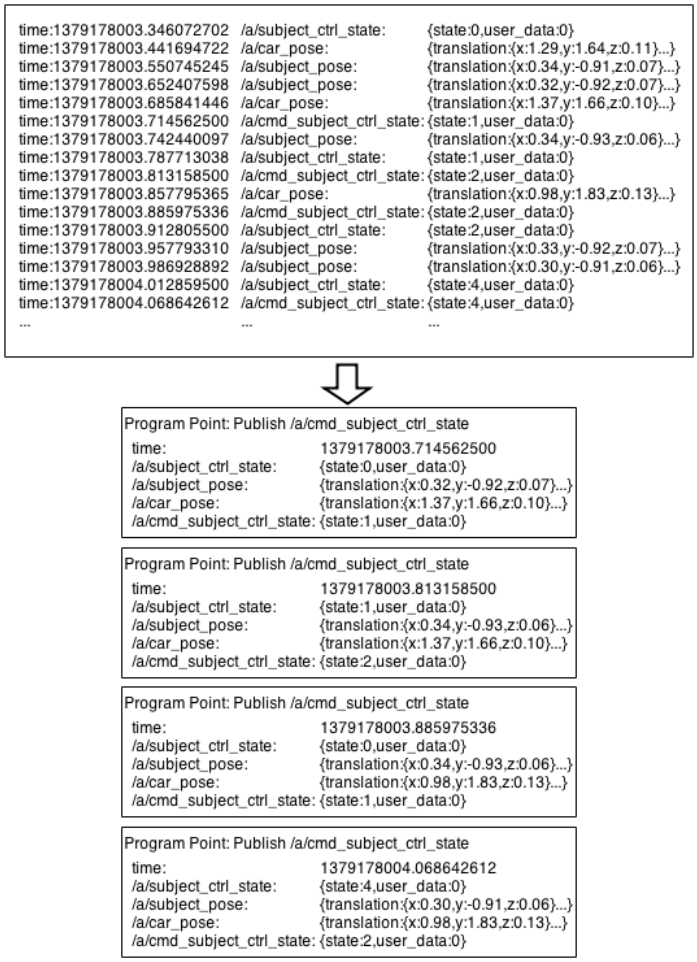

Figure 3.7: Sample trace (top) and message pairings (bottom)

the data trace at the architecture program points. Since Daikon does not support user defined data structures, we represent the architecture data as string variables as shown in Figure 3.8.

Our approach can also infer temporal invariants, which treats publishing a message or calling a service as an event. Since all these events have been recorded in bag files and they are ordered and stamped by time, the translator only needs to retrieve the events indexed with their time stamps as shown in Figure 3.9. In this example, the user chooses three events (*/a/task_waypose*,

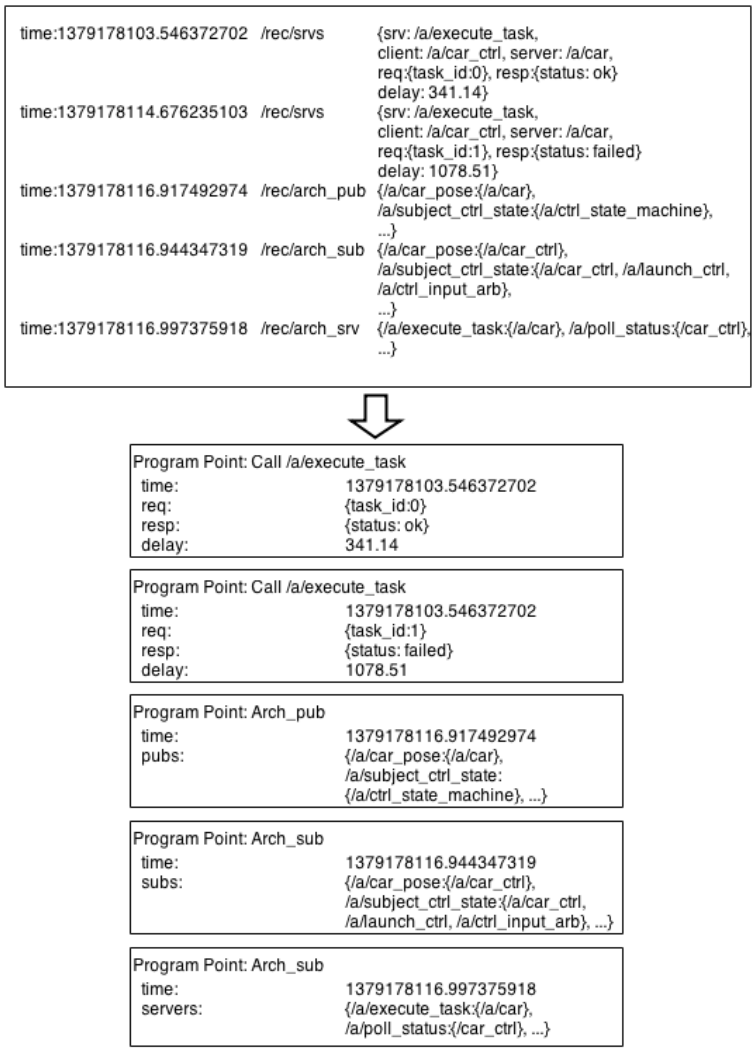

Figure 3.8: Sample services and architecture trace (top) and their translations (bottom)

/a/cmd_subject_ctrl_state, */a/execute_task*) to infer temporal invariants, so the translator extracts these three events from bags files and orders them by their time stamps.

The translator conducts all the jobs under the instructions from the configuration file, where the *scope* tag declare the program points in terms of topics, services, architectures and temporal properties. Figure 3.3 shows a sample configuration file , where two topics are declared in the *publish* tags: */a/cmd_subject_ctrl_state* and */a/task_waypose*. The attribute *relative* defines how the translator will group messages to fill the program points of Daikon's input. If *relative* is set with 1, the translator will group the input and output topics with one publisher as previous discussed, but it can be set to group messages across multiple publishers.

Monitoring services are defined in the *call* tag, as the service */a/execute_task* is declared in the sample configuration in Figure 3.3. Monitoring architectures is set in the *arch* tag, where the user can declare to analyze publishers, subscribers, or services architecture individually by changing its contents. The *temporal* tag declares a temporal analysis with an event scope defined in the *events* attribute. For example, in Figure 3.3 the *events* attribute refers to three events by their labels: a. publish to */a/cmd_subject_ctrl_state*, b. publish to */a/task_waypose* and c. call service */a/execute_task*. The task of the translator is to retrieve these events from the bag files and to put them into a temporal program point in Daikon as shown in Figure 3.9. Since there is one *temporal* element in the configuration file, there is only one temporal program point named *temporal+element_id*. Each instance of the program point contains one string variable consisting of the event and its time stamp.

The overall translation procedure is shown in Algorithm 1. It first parses the configuration file to initialize the *msgTable* and *programPointTable*. The *msgTable*

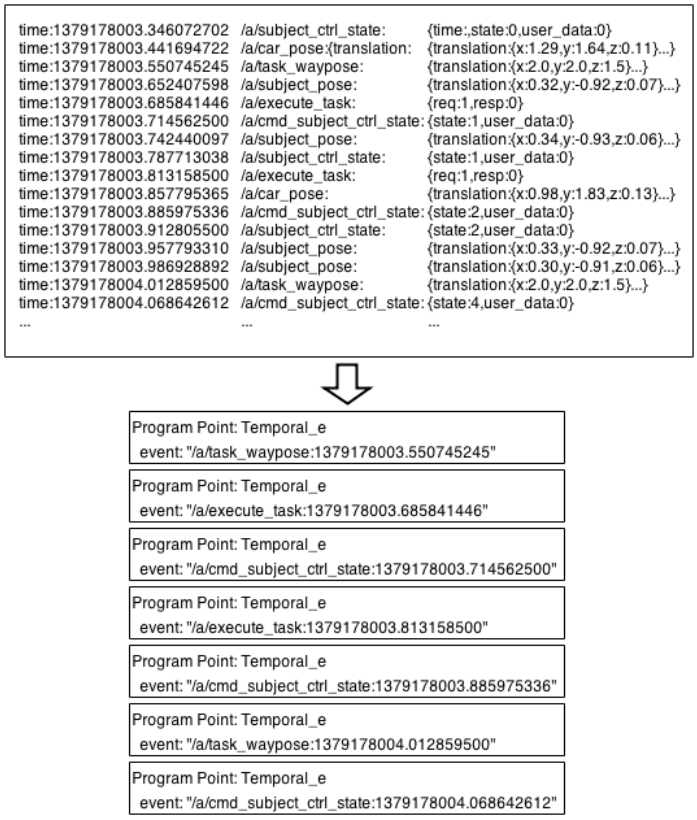

Figure 3.9: Sample trace (top) and event sequence (bottom)

maps a topic name to a message instance, and the *programPointTable* maps a topic name to its corresponding program point which is a list of message instances in the *msgTable*. And then it iterates through all the messages in the bag. If the message table contains the message's topic, it updates the message value in the table. If the message's topic is a key in the *programPointTable*, it outputs the program point instance into the output trace. ROS's rosbag tool provides a C++ API, so we implemented a C++ version of the translator, which reads the bag files and the configuration file, and outputs the data trace files in Daikon's format, including a *.decl* file and one or more *.dtrace* file(s).

Algorithm 1 Translation Procedure

```
1: parse config
2: initialize msgTable
3: initialize programPointTable
4: while msg in bag do
5:    if msgTable contains msg.topic then
6:       update msgTable
7:    end if
8:    if programPointTable contains msg.topic then
9:       output programPointTable[msg.topic]
10:   end if
11: end while
```

3.4 Inferring Invariants

Techniques that infer invariants from program executions often target a set of standard invariants such as the memory locations read or written at marked program points [28] or the ranges of values observed for a variable at the entry or exit points of a function [3]. Daikon provides default invariant templates including unary, binary and ternary ones on scalar, string and array values. For arrays of scalars, we implemented two additional invariant templates that capture the ranges of the average and the standard deviation of the arrays, because it is useful to characterize array variables representing, for example, sensors values by these two properties. Moreover, identifying richer invariants, like the ones we aim to capture in robotic systems, requires the specification of richer invariant templates. Through this work we introduce four new types of invariant templates shown in Table3.1 that reflect the spatial, time, architectural, and temporal nature of robotics systems.

Invariant	Description	Example
Time-related	Messages' frequency, variance and change rate	$freq(m1) > 21$ $var(m1.a) < 2.3$ $rate(m1.a) < 1.3$
Polygon	Relationships between two variables (2-D range)	$\bigcup_i^n (a_i x + b_i y + c[>= \mid <=]0)$
Architecture	Node-communication graph	$max\_pub(t1) = \{n1, n2\}$ $min\_sub(t1) = \{n3\}$
Temporal	Temporal properties of events' sequences	$interval\_time(e1, e2) < 0.1$ $interval\_max\_events(e1, e2) = \{2 \times e1\}$ $((e1\|e2)e3)^*$

Table 3.1: New Invariants Templates

3.4.1 Time-Related Invariants

First, we introduce a new type invariant that incorporates time as a central component. In robotic systems the program state is not just presented in variables' values, but also in time-related variable values such as frequency, variance or change rate. The simplest of these templates serves to characterize the messages' frequency and variance. For frequency, this takes the form of $constantLower \leq message\_frequency \leq constantUpper$. For our sample system introduced in Section 5.1 and Figure 1.1, this type of invariant is useful to detect, for example, stale location data that may direct the UAV to the wrong location when the communication is broken.

The variance invariant takes a similar form as $constantLower \leq variable\_variance \leq constantUpper$. This type of invariant can also detect some stale data from a broken sensor. For example, usually the sensor gives data with some variance. If it is broken and gives stale data, the variance invariant will detect that error.

A more complex type of invariant aims to capture the derivative of continuous raw variables. For example, the derivatives of distance traveled over time may render velocity or acceleration invariants. This type of invariant also takes the form

of $constantLower \leq variable\_rate \leq constantUpper$. In our scenario, a common instance of such invariant of this type is $minVelocityUAV \leq VelocityUAV \leq maxVelocityUAV$, which can detect wrong localization data as shown in our case study in Section 5.1.

To infer these invariants, we take advantage of the time stamps attached with the observation at the program points, and make three new invariant templates in Daikon to associate the time component with variables at the same program point. To filter out short-time noise (the interval of two messages is very small which makes the change rate extremely large), the templates do the computation in a time window instead of on every consecutive message pair. The updating invariant procedures of the three new templates are shown in Algorithm 2, 3 and 4. The input of the algorithms is a trace file. Every time the algorithms are fed a timestamped record with a variable and a value, they update the queues within the time windows (the time difference between the first and the last elements in the queue should be less than the window size), and then compute their derived values (frequency, variance and change rate) that will serve to instantiate the corresponding invariants. To compute frequency, Algorithm 2 only needs the time, while to compute variance and change rate Algorithm 3 and 4 need the time and the target variable. And Algorithm 3 works on all elements in the queue to get the variable variance, while Algorithm 4 only computes the change rate based on the first and the last elements in the queue. Note that Daikon takes care of associating the time variable with other variables at the same program point and feeds them to the invariant templates. The time window can also be tuned in the configuration file.

Based on the algorithm, we can see the time complexity of Algorithm 2 and 4 is $O(N)$, where N is the length of the data trace. The time complexity of Algorithm 3

Algorithm 2 Frequency Invariant Inference ($trace, window$)

$freq\_max \leftarrow 0.0$
$freq\_min \leftarrow Float.max\_value$
$queue \leftarrow newQueue()$
while $time$ in $trace$ **do**
  $queue.enqueue(time)$
  while $queue.last() - queue.first() > window$ **do**
    $queue.dequeue()$
  end while
  $freq \leftarrow \frac{queue.size()}{window}$
  $freq\_max \leftarrow max(freq, freq\_max)$
  $freq\_min \leftarrow min(freq, freq\_min)$
end while
output $freq\_max$ and $freq\_min$ as invariants

Algorithm 3 Variance Invariant Inference($trace, window$)

$var\_max \leftarrow 0.0$
$var\_min \leftarrow Float.max\_value$
$value\_queue \leftarrow newQueue()$
$time\_queue \leftarrow newQueue()$
while $value$ and $time$ in $trace$ **do**
  $value\_queue.enqueue(value)$
  $time\_queue.enqueue(time)$
  while $time\_queue.last() - time\_queue.first() > window$ **do**
    $time\_queue.dequeue()$
    $value\_queue.dequeue()$
  end while
  $variance \leftarrow Statistic.variance(value\_queue.toList())$
  $var\_max \leftarrow max(variance, var\_max)$
  $var\_min \leftarrow min(variance, var\_min)$
end while
output var_max and var_min as invariants

Algorithm 4 Rate Invariant Inference($trace, window$)

```
1: rate_max ← Float.min_value
2: rate_min ← Float.max_value
3: value_queue ← newQueue()
4: time_queue ← newQueue()
5: while value and time in trace do
6:    value_queue.enqueue(value)
7:    time_queue.enqueue(time)
8:    while time_queue.last() − time_queue.first() > window do
9:       time_queue.dequeue()
10:      value_queue.dequeue()
11:   end while
12:   if value_queue.size() > 1 then
13:      rate ← (value_queue.last() − value_queue.first()) / (time_queue.last() − time_queue.first())
14:      rate_max ← max(rate, rate_max
15:      rate_min ← min(rate, rate_max)
16:   end if
17: end while
18: output rate_max and rate_min as invariants
```

is $O(N * q)$, where q is the queue size that depends on the window size. Usually, we use a small window size, so q is much smaller than the length of data trace N, so it is $O(N)$.

3.4.2 Polygon Invariants

We introduce invariant templates that define relationships between two variables[1] that can be characterized through a convex polygon. This type of invariant is valuable to capture physical space bounds. For example, if our operating scenario was bounded by the dimensions of a room, this invariant template would be instantiated and refined into a polygon similar to the shape of the room. This

[1]We note that we did explore invariant templates with more than two variables and although some of the instantiated invariants were useful, we found that the cost of invariant generation was exponential and hence prohibitive unless it was focused on a small set of topics.

type of invariant can also characterize relationships between variables that are hard to anticipate because of their lack of linearity. Take the UAV acceleration and its pitch and roll for example. Ideally, these variables are linearly correlated. However, wind velocity may introduce variation in these relationships that can only be captured through the richer invariants like the ones we are proposing.

This invariant template takes the form of $\cap_i^n(a_i x + b_i y + c[>= \mid <=]0)$ that defines a polygon of n sides. Every time a new variable-value is read from a trace, it is checked against the polygon. If it resides inside the polygon, it is ignored. If it resides outside the polygon, the polygon is relaxed by computing the convex hull that includes the new observation.

We implemented a new polygon invariant template in Daikon, where we use a basic divide-and-conquer quickhull algorithm[17] as shown in Algorithm 5. The input of quickhull algorithm is a set of points, which is provided by Daikon who pairs two input variables and generates a set of values' pairs as the point set. Then the algorithm first computes a line between two extreme points in one direction (X axis for example). Based on the side the point locates at, it separates the point set into two sets ($upper\_set$ and $lower\_set$, see Lines 1-2 in Algorithm 5), then for each set it finds the one-side contours recursively (see $Function\ find\_lines$ in Algorithm 5), and finally it composes the polygon with the computed contours. Its average case time complexity is $O(N * logN)$. Ideally, for each pair of values, we can first store them in a point set, and at the end we can run this algorithm only once on the entire point set. However, we may run out of memory in this way, because it needs to store a huge amount of values pairs from the data trace. So, we chop the data trace into chunks and run the algorithm on them one by one as show in Algorithm 6. Before adding point into chunk, it first checks if the point is inside of the computed polygon. It only adds the point outside of the polygon into

the chunk to update the polygon. This does not affect the outcome of the process, but it mitigates the memory consumption at the cost of performance. We carried out a performance test in our first case study (Sec 5.1), where we only inferred polygon invariants on the same data trace with different chunk sizes. The result in Figure 3.10 shows that the performance degrades if we make the chunk size too big. We empirically choose 20 as a fixed chunk size in our implementation.

Algorithm 5 quickhull(*point_set*)

```
1: find point_min_x and point_max_x
2: divide point_set into point_upper_set and point_lower_set
3: line ← (point_min_x, point_max_x)
4: line_upper_list ← find_lines(line, point_upper_set)
5: line ← (point_max_x, point_min_x)
6: line_lower_list ← find_lines(line, point_lower_set)
7: return line_upper_list + line_lower_list
8:
9: Function find_lines(line, point_set)
10:    make empty line_list
11:    if (point_set is empty)
12:       add line into line_list
13:    else
14:       find the point point with the maximum distance from line
15:       line1 ← (line.p1, point)
16:       pull points outside of the line from point_set to point_set1
17:       line_list_1 ← find_lines(line1, point_set1)
18:       add all line_list_1 into line_list
19:       line2 ← (point, line.p2)
20:       pull points outside of the line from point_set to point_set2
21:       line_list_2 ← find_lines(line2, point_set2)
22:       add all line_list_2 into line_list
23:    end if
24:    return line_list
25: EndFunction
```

Algorithm 6 Polygon Invariant Inference(*trace*)

```
1: initialize polygon and point_set
2: while v1 and v2 in trace do
3:    point ← newPoint(v1, v2)
4:    if polygon.not_contains(point) then
5:       add point into point_set
6:    end if
7:    if point_set.size() ≥ chunk_size then
8:       add all points in polygon into point_set
9:       polygon ← quickhull(point_set)
10:      clear point_set
11:   end if
12: end while
13: add all points in polygon into point_set
14: polygon ← quickhull(point_set)
15: output polygon as polygon invariant
```

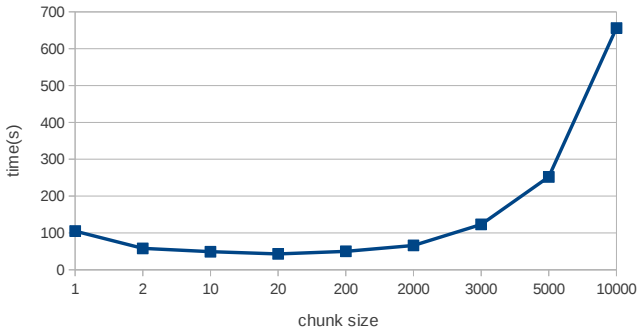

Figure 3.10: Process Time vs. Chunk Size

3.4.3 Architecture Invariant

Distributed robotic systems have a dynamic architecture that can be tweaked for different deployment conditions. These tweaks can often lead to erroneous conditions, causing additional or missing topics or nodes.Take the ROS program shown in Figure 3.11 for example. The correct architecture is shown with the solid lines, where the node */a/car* publishes to the topic */a/car_pose* and */a/UAV* publishes to */a/subject_pose*. Since the two nodes */a/car* and */a/subject* are spawn from the same source code but with different remapped names, it is easy

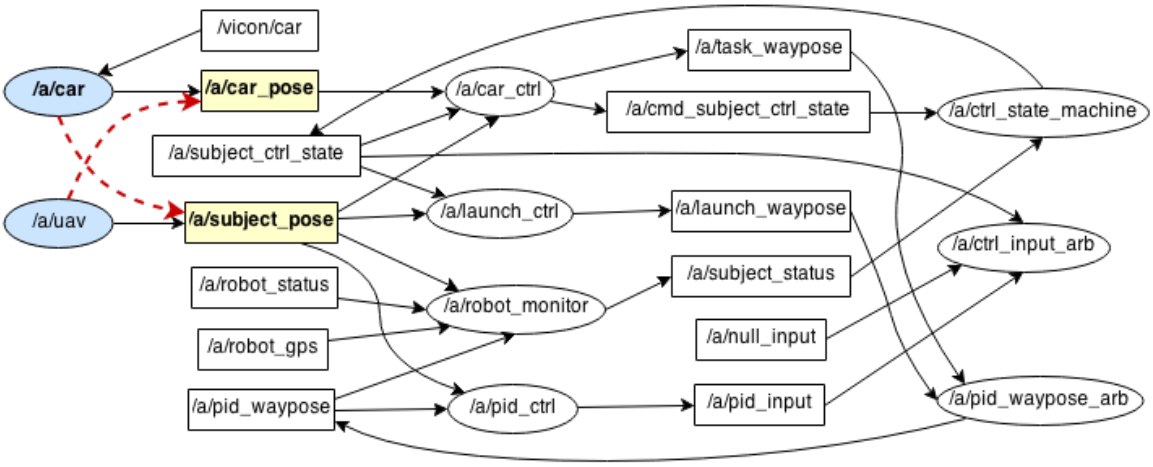

Figure 3.11: A ROS Program Example

for users to create mappings that cause incorrect connections as shown with the two dashed lines in Figure 3.11. Although the messages' values may look correct, the system is in a dangerous state, because the computations are based on the wrong messages, and the system may generate wrong control inputs and crash the robot.

Our approach can capture architecture invariants for two kinds of resources: topic and service, which are the two main mechanisms for nodes to communicate in ROS. For each topic, we have a set of nodes as publishers and also a set of nodes as subscribers as shown in Figure 3.11. For each service, we only have one server at a time, but there might be multiple servers providing it at different times.

The architecture invariants take the form of the maximum and minimum nodes set using some particular communication resources as
$\{resource : (max\_node\_set, min\_node\_set), ...\}$.
For example, Figure 3.11 shows the graph for nodes communicated through topics. If the architecture does not change, this architecture is an invariant for the topics. For the topic $/a/subject\_pose$ topic, the invariant of publishers is $/a/subject\_pose$:

$(\{/a/uav\}, \{/a/uav\})$, which means there must be only one node named $/a/uav$ publishing to this topic. If the system is launched with incorrect mappings as shown in dashed lines, the architecture invariant will be violated, because $/a/uav$ does not publish to topic $/a/subject\_pose$ or $/a/car$ publishes to this topic. For subscribers of the topic $/a/subject\_pose$, the invariant is $/a/subject\_pose$: $(\{/a/car\_ctrl, /a/launch\_ctrl, /a/pid\_ctrl\}, \{/a/car\_ctrl, /a/launch\_ctrl, /a/pid\_ctrl\})$. If $/a/car\_ctrl$ is killed for some reason, the invariant will also be violated due to the unexpected architecture.

We have developed a new invariant template in Daikon, which infers the maximum and minimum set of publishers, subscribers, and servers from architecture variables in strings. As shown in Algorithm 7, given an architecture string, the update procedure first parses the string to get the resource type (publishers, subscribers or servers), and then extracts the nodes into a node set. Next, it updates the corresponding maximum and minimum sets. The time complexity is $O(N * k)$, where k is the size of the maximum node set. Since k is much smaller than N, the time complexity is also N. An example of such architecture invariants outputted by Daikon is shown in Figure 3.12, where the three architecture for publishers, subscribers, and servers are specified respectively, and for each architecture the maximum and minimum sets are presented as map variables.

3.4.4 Temporal Invariant

Another category of invariant is the temporal invariant, which expresses order properties of events' sequences. For example, in a multi-thread program, usually to access a critical variable, the program needs to get the lock first and then release the lock after finishing accessing the variable. The lock and unlock events should

Algorithm 7 Architecture Invariant Inference(*trace*)

1: **while** *arch* in *trace* **do**
2: parse *arch* into *node_set*
3: **if** not initialize *max_set* **then**
4: $max\_set \leftarrow node\_set$
5: $min\_set \leftarrow node\_set$
6: **else**
7: $max\_set \leftarrow max\_set \bigcup node\_set$
8: $min\_set \leftarrow min\_set \bigcap node\_set$
9: **end if**
10: **end while**
11: output *max_set* and *min_set* as invariant

```
================================================================================
Arch_pub:::POINT
max_pubs:{/a/car_pose:[/a/car],/a/subject_ctrl_state:[/a/ctrl_state_machine],
/a/cmd_subject_ctrl_state:[/a/car_ctrl],/a/subject_status:[/a/robot_monitor]}
min_pubs:{/a/car_pose:[],/a/subject_ctrl_state:[/a/ctrl_state_machine],
/a/cmd_subject_ctrl_state:[],/a/subject_status:[/a/robot_monitor]}
================================================================================
Arch_sub:::POINT
max_subs:{/a/car_pose:[/a/car_ctrl],/a/subject_ctrl_state:[/a/car_ctrl,/a/launch_ctrl,/a/ctrl_input_arb],
/a/cmd_subject_ctrl_state:[/a/ctrl_state_machine],/a/subject_status:[/a/ctrl_state_machine]}
min_subs:{/a/car_pose:[],/a/subject_ctrl_state:[/a/car_ctrl],/a/cmd_subject_ctrl_state:[/a/ctrl_state_machine],
/a/subject_status:[/a/ctrl_state_machine]}
================================================================================
Arch_srv:::POINT
max_srvs:{/a/execute_task:[/a/car],/a/poll_status:[/a/car_ctrl,/a/robot_monitor]}
min_srvs:{/a/execute_task:[/a/car],/a/poll_status:[/a/robot_monitor]}
```

Figure 3.12: Architecture Invariants

always happen in the correct order *lock* $\rightarrow$ *unlock*. In our approach we focus on two kinds of events: publish to a topic and call to a service, so the invariants will define temporal orders or patterns between them. In our UAV system, we also observed some temporal properties. For example, the *pid_ctrl* node should not publish control messages until it gets the iRobot and the UAV position messages. And if the *pid_ctrl* node does not receive position messages for a while, it should stop publishing control messages.

We have two temporal invariant inference templates: ordered-pair interval and pattern. The order-pair interval invariant captures a common pattern in robotic systems, like a landing event is always followed by decreasing the thrust

or a moving forward event should always follow a pitch command. And the pattern invariant detects specific temporal pattern specified by the user that can be represented by simple regular expressions.

3.4.4.1 Ordered-Paired Interval

The ordered-pair interval invariant captures the properties about the intervals of ordered event pairs in event sequences. It expresses that an event is followed by another event as shown in the pattern $(\bar{A}^*A\bar{B}^*B)^+$, where A is followed by B. It also specifies the interval information between the first event and the second event. The interval information includes time and events, so the invariants capture maximum/minimum interval time and maximum/minimum events happening in the interval. The maximum/minimum events are multi-sets of events. However, if the second event happens too far away from the first event in terms of the time or the number of events, we would not consider it as an ordered-pair, because we interpret it as a weak pair relationship. This invariant takes the form of $\cap(e1 \rightarrow e2 : \{max\_time, min\_time, max\_events, min\_events\})$.

As an example shown in Figure 3.13, the first column is an event trace of three events a,b,c and their time occurrence. We first analyze the ordered-pair interval of $a \rightarrow b$, where the inference engine first initializes four interval instances in the second column, and then infers the interval invariants as shown at the bottom of this column. The first two are the maximum and minimum interval, which indicates that once a happens b should happen within 1 to 2 seconds. And the maximum and minimum event sets in the interval are empty, which means b should follow a without any other events (a or c) happening in the interval. So the event sequence like acb or aab would violate these invariants. From the same trace, it can infer the interval invariant of $a \rightarrow c$ as shown at the bottom of the third

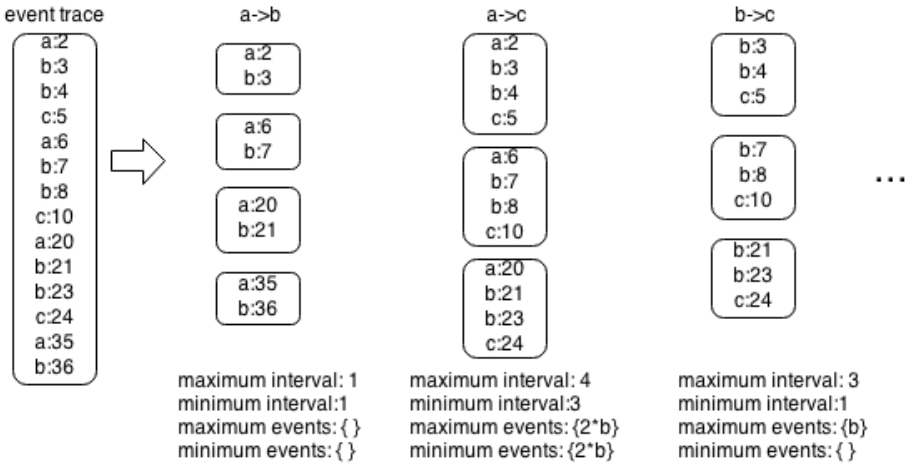

Figure 3.13: Event Trace and Interval Analysis

column. The maximum and minimum event sets are both $\{2 * b\}$ with means that, every time we see a, we would expect exactly two b events happen before an event c happens. If we see one b or three b or one a before c, the invariant is violated. The fourth column is the interval analysis for $b \rightarrow c$. The minimum event set is empty and the maximum event set is $\{b\}$, which indicates that, if b happens, one another b event may happen or nothing may happen before the event c happens. If some event beyond the multiset happens, the invariant is violated. On the event scope of a, b, c, three other intervals invariant ($b \rightarrow a$, $c \rightarrow b$ and $c \rightarrow a$) would be generated also.

We implemented a new template for the ordered-pair interval invariant in Daikon, as shown in Algorithm 8. Based on the event scope (see *events* attribute in *temporal* tag in Figure 3.3) defined in the configuration file, it first initializes all the ordered events pairs. Then, upon an event from the event trace, it updates all the ordered-pair intervals (see Line 3-25 in Algorithm 8). If the interval time is larger than a threshold (predefined to 2 seconds but configurable) or the size of the event set is greater than a threshold (predefined to 10 events but configurable),

it removes this interval (see Line 19-22 in Algorithm 8). Finally, it outputs all the ordered-pair interval invariants that happens in the trace (see Line 27-31 in Algorithm 8). The time complexity of this inference is $O(N * k^r * s)$, where k is the number of events in the event scope, r is the number of symbols in the regular expression, and s is the number of state in the DFA machine generated from the regular expression.

3.4.4.2 User Defined Pattern

The second temporal invariant template targets a pattern defined by the user in the form of a regular expression, which is declared in the *pattern* tag in the configuration file. The invariants inferred will also be regular expressions as $\bigcup regular\_exp(events)$. For instance, in Figure 3.3 the user declares a pattern $(AB)^*$ with the events scope:

a. publish to $/a/cmd\_subject\_ctrl\_state$,

b. publish to $/a/task\_waypose$ and

c. call service $/a/execute\_task$.

The inference engine will try to infer concrete patterns with these events. On the event trace shown in Figure 3.9, the inference engine will find two invariants:

$((/a/cmd\_subject\_ctrl\_state)(/a/execute\_task))^*$ and

$((/a/execute\_task)(/a/cmd\_subject\_ctrl\_state))^*$.

As a result, this invariant template is useful when the user has the domain expertise to identify the kinds of patterns that may occur.

The pattern inference process works as shown in Algorithm 9. First, based on a regular expression(such as $(AB)^*$), it generates a DFA on the symbols (such as A, B). Second, according to the event scope (such as $/a/cmd\_subject\_ctrl\_state$, $/a/task\_waypose$ and $/a/execute\_task$), it finds all the permutations of the events

Algorithm 8 Ordered-Pair Interval Invariant Inference(*trace*)

```
1: initialize interval_list on the event scope
2: while event in trace do
3:    for interval in interval_list do
4:       if interval.isClose then
5:          if event is interval.first then
6:             interval.isClose ← false
7:          end if
8:       else
9:          if event is interval.second then
10:            if interval.time > time_threshold then
11:               remove interval
12:               break
13:            end if
14:            update interval.max_time and interval.min_time
15:            update interval.max_events and interval.min_events
16:            interval.isClose ← true
17:         else
18:            add event into interval.current_events
19:            if interval.time > time_threshold or interval.current_events.size >
               events_threshold then
20:               remove interval
21:               break
22:            end if
23:         end if
24:      end if
25:   end for
26: end while
27: for interval in interval_list do
28:    if interval has been initialized then
29:       output interval
30:    end if
31: end for
```

and then uses these permutations to generate all the DFA machines with transitions on concrete events(such as $((/a/cmd\_subject\_ctrl\_state)(/a/execute\_task))^*$). Since the monitoring of the property can start at any point during the program execution, each state machine has multiple current state pointers which are initialized pointing to all its states. Third, on each event, the algorithm drives all the DFA machines (each pointer goes to its next state on that event), and kills the machine if all the pointers go into error states, since that indicates that there is no way for the regular expression to hold. Finally, for each DFA machine, if there is one pointer that has traversed all its states, it outputs this DFA machine in the form of a regular expression on its concrete events.

Algorithm 9 Pattern Invariant Inference

```
1: parse pattern into a general_DFA
2: from general_DFA generate concrete DFA_list on the event scope
3: while event do
4:    for DFA in DFA_list do
5:       drive DFA on event if applicable
6:       remove DFA if all the pointers go into error
7:    end for
8: end while
9: for DFA in DFA_list do
10:   if all the states in DFA has been traversed by its one pointer then
11:      output DFA
12:   end if
13: end for
```

We use multiple pointers as the DFA's current state, because the recorded event traces may not start from the very beginning of the executions, so the state machine may not be at its starting state at the beginning of the event trace. As shown in Figure 3.14, the rosbag tool may be launched later than the system, so the recorded trace starts with "ABAAB" rather than "AABAAB". As a result, we cannot find the pattern (AAB)* with a normal DFA. So we initialize the DFA machine with a state

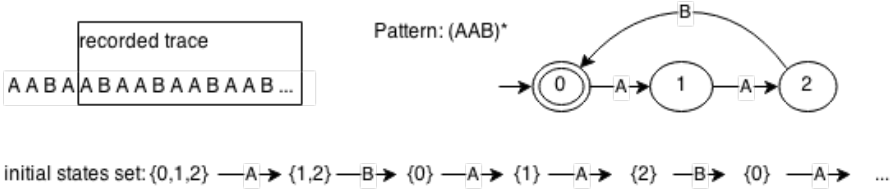

Figure 3.14: State Set Transition of a Sample DFA

pointer set pointing to all its states, and at each input drive each state in the set into its next state, and remove it if it goes into error state, as shown in Figure 3.14. So that we can find the pattern from the partial event trace. In the last step, the engine checks the traversed states of each pointer to verify the state machine, and it only outputs the machines with good evidence.

The time complexity of the algorithm is $O(N * k^2)$, where k is the number of events in the event scope.

The regular operations we support includes *Union* or *Or* (as "|"), *asterisk* (as "$*$"), and *Grouping* or *Parentheses* (as "()" can be nested). The alphabet set is limited to 26 alphabet letters (as $a..z$ and $A..Z$) and *Epsilon* as ("$-$").

3.5 Invariants Justification

At the end of the inference process, invariants that are statistically justified are outputted. For the default invariant inference templates, Daikon provides the mechanism for invariants justification, which computes a confidence level to filter out invariants that are satisfied purely by chance. Each type of invariant has its own rules for determining confidence, as defined in the $computeConfidence$ method in the invariant template. The confidence computation is between 0 and 1, which relates to the number of samples that satisfy the invariant. For example,

consider the Daikon's default confidence computation is $1 - 1/2^{num\_samples}$. With 3 samples, the invariant's confidence is 0.875, and the invariant will be suppressed by the default confidence limit 0.99. If the confidence level for the invariant is larger than the limit (7 samples), then Daikon outputs the invariant.

All new invariant templates apply the same statistical mechanism to compute the confidence level. Each time a new data is analyzed, it checks its invariants. If the invariant needs to be updated by the new data, the number of samples goes back to 1; otherwise, it increases the sample size by 1. At the end, Daikon computes the confidence level by $1 - 1/2^{num\_samples}$ and decides whether to output the invariant.

For our polygon invariant template, we use a different justification method by comparing the polygon areas. The idea is that if two stochastic variables are independent from each other, then, with enough samples, the polygon boundaries will approximate a rectangle as shown in Figure 3.15. In that case, we can drop the polygon invariant because it provides almost the same constraints as the boundary invariants on the two individual variables. By dropping the invariant, we reduce the overhead of checking the polygon invariant without sacrificing much precision. In our approach, we measure the area ratio of the polygon over the rectangle to decide whether to drop it as shown in Algorithm 10. The time complexity of the algorithm is $O(k)$, where k is the number of vertexes of the polygon. The threshold of dropping can be set in the configuration file.

We performed an experiment using the data from the "normal" scenario of our first case study (Section 5.1), which is a system designed to land a UAV on a moving platform. We generated different sets of polygon invariants with different dropping ratios, and used them to check 76 failed runs. If any invariant was broken in the check, we investigate whether this set of invariants can detect the

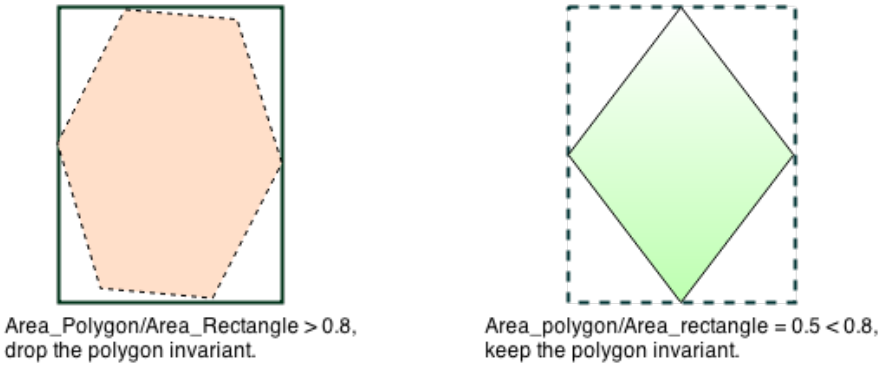

Figure 3.15: Polygon Justification

Algorithm 10 Polygon invariant compute confidence

1: compute *area_polygon*
2: compute *area_rectangle*
3: $ratio \leftarrow \frac{area\_polygon}{area\_rectangle}$
4: **if** $ratio > threshold$ **then**
5: $\quad confidence \leftarrow 0$
6: **else**
7: $\quad confidence \leftarrow 1$
8: **end if**

faults in the test cases. The result is shown in Table 3.2. We can see that, with smaller dropping ratios, the number of polygon invariants and polygon sides decreases without degrading the failure detection (at least for the faults in the 76 failed runs). The column *violated polygons* indicates that almost all polygon invariants were ever violated in the 76 failed runs. The *Failed Runs with Polygon invariant violations* is almost 100%, which could be resulted from that in the 76 failed runs almost all the failures could be detected by the range invariants, where the polygon invariants (2-D range) put more restrictive constraints than the default (1-D range) invariants.

Dropping Ratio	Invariants	Polygons	Sides	Violated Polygons	(%) Failed Runs with Polygon invariant violations
1.0	755	466	6122	465	100(76/76)
0.9	702	413	5107	413	100(76/76)
0.8	644	355	4145	355	100(76/76)
0.7	568	279	3078	279	98.7(75/76)
0.6	429	140	1494	140	98.7(75/76)
0.5	329	40	524	40	97.4(74/76)

Table 3.2: Evaluation of Dropping Polygons

3.6 Conditional Invariants

We introduce the notion of conditional invariants, that is, invariants that can only hold under certain conditions that can be identified as such. For example, in our system the invariants that hold when the UAV is on the ground versus when it is flying are quite different. This partition of the space of system behavior helps to generate more and more precise invariants for subsets of the system states. Attempting to produce invariants without differentiating such states would result in a smaller set of more general invariants, but it would miss many valuable invariants that only apply to one system state. For example, the critical invariants that characterize how the system should behave when attempting to land on the moving platform (e.g. the UAV and platform X and Y coordinates should be within a certain threshold) would be dropped, as they would not hold when the UAV is pursuing the landing platform.

To infer such invariants, we heuristically use the composition of existing invariants templates. First, we run Daikon on the whole data traces and identify variables that have a small discrete set of values, which are variables with less than 10 values (configurable parameter). This helps us identify variables such as $UAVmodes$ which has a range from 0 to 8 indicating whether the UAV is taking off, hovering, translating, landing, etc. Second, we partition the traces into

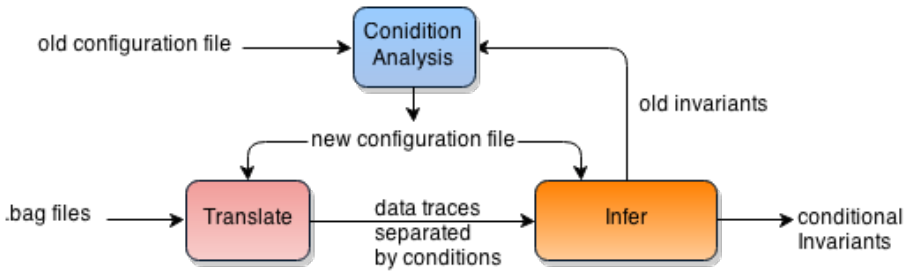

Figure 3.16: Conditional Invariant Inference

sub-traces according to those discrete values. Third, we perform inference on the sub-traces independently and incorporate the learned invariants and a predicate on the discrete variable as part of the monitor. This heuristic approach identified the state variable correctly in our case study one, but in some cases it may find multiple such variables and divide the whole data trace into too many small sub traces. Or if the state variable takes more than the specified values, this approach would fail to find such variable. Given the importance of conditional variables, we expect that in practice the software engineer will have to annotate such variables or, as we have done in our studies, manually check them.

The workflow is shown in Figure 3.16, the *ConditionAnalysis* component parses the invariants, and searches for variables with a small discrete set of values, which is an invariant in Daikon in the form of "one of {...}". Based on the searched variables and its program point, it generates a new configuration file, which is injected with the conditions and corresponding conditional program points. Then, under the new configuration, the translator generates data traces separated by conditions on conditional program points. And finally Daikon outputs conditional invariants.

Inferring conditional invariants adds conditional program points, where each program point has a sub trace of original traces. The size of the data traces to be

processed is the same as before, so the cost should be equivalent except for the additional analysis.

The user can also make the conditional analysis effective by declaring the condition inferences in the configuration file by adding desired condition tags as shown in Figure 3.3.

3.7 Invariant Inference Summary

In this chapter, we have described our invariant inference on ROS-base robotic systems. The whole process has three steps: trace generation, trace translation and invariant inference.

In the first trace generation step, we take advantage of rosbag tool to record all messages on topics, and add a recording node to record ROS services, architectures and parameters. This approach allows us to capture all necessary data by building on the strengths of the existing ROS toolset.

In the trace translation step, the translator clusters topic messages according to their publishing/subscribing relations, and utilizing the information in a configuration file, it puts topic, service and architecture messages into their corresponding program points in Daikon's format.

In the invariant inference step, we extend the invariant templates with four new kinds of invariants: time-related, polygon, architecture, and temporal, which capture some critical properties in robotic systems.

Chapter 4

Monitor Synthesis

In this chapter, we will describe the monitor synthesis process in detail. The input to this procedure is a set of invariants, and the output is a monitor node which will be integrated into the system to check the inferred invariants automatically. Since most of the invariants are simply boolean expressions, it is straightforward to encode these invariants as predicates at the corresponding program points. However, to enhance the effectiveness of the synthesized monitor we explored two additional activities: invariant classification and recovery actions.

4.1 Monitor Synthesis Workflow

As shown in Figure 4.1, the workflow is separated into two steps: *InvariantClassification* and *MonitorSynthesis*. For invariant classification, we worked on refining the inferred invariant set. The inputs are an invariant set and extra system runs including successful and failed ones. The output is a refined invariant set with the same format as the input, so that this step can be injected or removed without affecting the next step. The main idea of this step is evaluating invariants as binary classi-

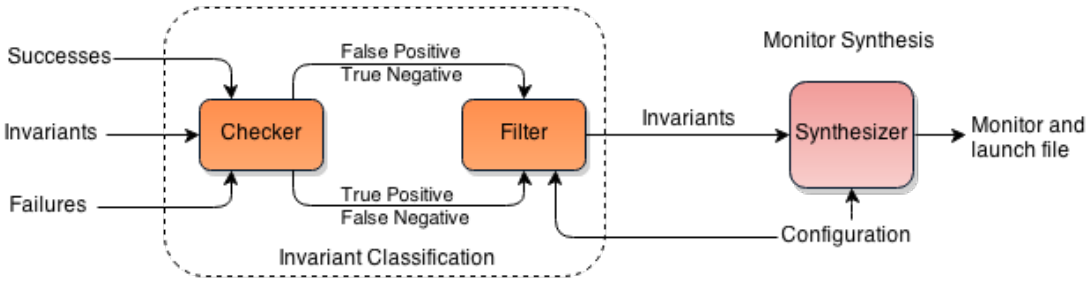

Figure 4.1: Monitor Synthesis Work Flow

fiers of system's state. With the help of successful and failed runs in bag files, the *Checker* first computes $FalseNegative(FN)$, $TruePositive(TP)$, $TrueNegative(TN)$ and $FalsePositive(FP)$ counts. Then, the *Filter* computes a F-score for each invariant, and based on a threshold defined in the configuration file it decides whether or not to keep the invariant. For monitor synthesis, besides checking the invariants at each program point, we inject actions to recover from detected invariant violations. These recovery actions range from default actions like raising a warning to some particular actions defined in the configuration file like blocking messages.

The two steps are set through the configuration file as shown in Figure 4.2 through the tags *check* and *monitor*. The tag *check* defines how to classify and filter the inferred invariants, where the element *success* gives the directory containing the additional successful runs while *fail* provides the failed runs, and the attribute *threshold* defines the threshold to filter out invariants. In the tag *monitor*, the recovery actions are defined and assigned to the monitored program points in the tag *scope*. The *block* tag specifies on which topics or services the message will be blocked. The *action* tag declares an action in the term of publishing a particular message to a topic, which are defined in the *topic* tag and *value* tag. The *violation* tag specifies what actions to take at violations of invariants at specific program points. More details and an example about these tags are provided in Section 4.3.2.

```
−<imROS project="demo">
  −<scope>
    −<publish id="a" relative="1">
       <topic> /a/cmd_subject_ctrl_state </topic>
     </publish>
    −<publish id="b" relative="1">
       <topic> /a/task_waypose </topic>
     </publish>
     <call id="c"> /a/execute_task </call>
     <arch id="d"> pub,sub,srv </arch>
    −<temporal id="e" events="a,b,c">
       <pattern> (AB)* </pattern>
     </temporal>
  </scope>
  −<detect inv="demo.inv">
     <bag> bags/demo </bag>
  </detect>
  −<check inv="checked.inv" threshold="0.8">
     <success> bags/succs </success>
     <fail> bags/fails </fail>
  </check>
−<monitor launch="demo.launch" inv="checked.inv">
     <block> a,b </block>
   −<action id="f">
       <topic> /a/cmd_subject_ctrl_state </topic>
       <value> state = 8 </value>
     </action>
     <violation> a,c,e -> f </violation>
  </monitor>
</imROS>
```

Figure 4.2: Monitor Part of a Configuration File

4.2 Invariant Classification

In this step, we aim to refine the inferred invariant set to be monitored by removing invariants that are not useful at detecting anomalies according to our evaluation. As shown in Figure 4.1, it consists of two components: *Checker* and *Filter*. *Checker* checks the input invariants against the recorded successful runs and failed runs, and outputs FN, TP, TN and FP for each invariant. The filter component computes the *F-score*s for all the invariants, and then prunes the invariants below the threshold defined in the *threshold* attribute in Figure 4.2.

There are at least two benefits from this procedure. First, it reduces the monitor overhead of checking , which may cause latency in relaying messages. For instance, in the case study in Section 5.1, checking 1206 polygon invariants per message has a 1.6 ms latency, while without invariant checking the latency is about 0.2 ms.

Secondly, because of insufficient samples of system's behaviors, some invariants over constrained the system, and hence too fragile. For example, as we shall see in Section 5.2, the polygon inference generates about n^2 polygons on a program point with n variables, and these polygons need lots of samples to make them stable. These invariants may also obfuscate meaningful invariant violations among the reported broken invariants.

To conduct the classification, we pick a small part (e.g. %10) from all successful runs. Although these successful runs can be used to further refine invariants, fragile invariants may still remain, and they will be easily violated with more runs. Instead of refining these fragile invariants, we can prune them by checking them on those successful runs. At the same time, we can also build confidence on the invariants which are not violated. We also take advantage of failed runs which may also happen in the training process. Failed runs can be used to measure the effectiveness of the invariant to detect errors. If an invariant is always violated in the presence of failures, it may characterize that kind of failure, and the monitor may need to prioritize it.

Since we use violations of invariants to detect anomalies in the system, each invariant is a binary classifier of system's state. So, we are trying to measure the performance of these classifiers or predictors. Intuitively, if an invariant is not violated in any successful runs but it is violated in all failed runs, then it is an ideal classifier with the highest performance to detect anomalies. On the other hand, an invariant that is broken in both the successful and failed runs is a poor classifier since it cannot distinguish among these different system states. In the analysis above, an ideal invariant holds only in successful system states while it is violated in abnormal system state. In practice, there will be probabilities instead of certainties.

As shown in Figure 4.1, using successful runs, we check each invariant to compute the true negative and false positive rates, while we get the true positive and false negative rates from failed runs. Equation 4.1 is used to compute the true negative of an invariant, which is the fraction of the successful runs that the invariant is not violated in all the successful runs. As shown in Equation 4.2 the false positive rate of an invariant is the fraction of the successful runs that the invariant is violated in all the successful runs. In the same way, we compute the true positive and the false negative of the invariant as shown in Equations 4.3 and 4.4. However, we need a single value to score each invariant, so we compute the *precision* and the *recall*, and label each invariant with the *F-score*. Precision is positive predictive value (PPV) computed by Equation 4.5, which means if the invariant is violated, how sure we can say the system is in dangerous state. Recall (also known as sensitivity) computed by Equation 4.6 is that if the system is actually in dangerous state how well the invariant indicates that. The *F-score* is a single value to evaluate invariants, which is computed by the harmonic mean of precision and recall (see Equation 4.7).

$$true\_negative = \frac{not\_violated\_runs}{successful\_runs} \tag{4.1}$$

$$false\_positive = \frac{violated\_runs}{successful\_runs} \tag{4.2}$$

$$true\_positive = \frac{violated\_runs}{failed\_runs} \tag{4.3}$$

$$false\_negative = \frac{not\_violated\_runs}{failed\_runs} \tag{4.4}$$

$$precision = \frac{true\_positive}{true\_positive + false\_positive} \tag{4.5}$$

$$recall = \frac{true\_positive}{true\_positive + false\_negative} \tag{4.6}$$

$$F_\beta = (1 + \beta^2) * \frac{precision * recall}{\beta^2 * precision + recall}, \text{ where } \beta \text{ is a constant coefficient} \tag{4.7}$$

Consider the sample data in Table 4.1. *Invariant*1 is an ideal invariant, because it has the highest *precision*, *recall* and *F-score*. *Invariant*2 is unlikely to happen in reality since we infer the invariants from successful runs. Among *Invariant*3, 4 and 5, *invariant*5 is the best one, since it has the highest precision and recall; *Invariant*4's poor recall lowers its *F-score*; and *invariant*3 is the poorest one, given its lowest precision and the fact that it can only detect a tenth of the failures. For *Invariant*5, 6 and 7, *invariant*5 is the best one; *Invariant*4's precision is lower than *Invariant*5, which make its *F-score* lower than *Invariant*5; and *Invariant*7 is poorest one among this three invariants for its lowest precision. We can adjust the value of β to give different priority to precision and recall. As shown in Table 4.1, $\beta = 1.0$ means giving equal weights to precision and recall, while a smaller β value gives priority to precision and a larger one gives priority to recall.

Invariant	Successes		Failures		precision	recall	F score		
	% true negative	% false positive	% true positive	% false negative			β0.5	β1.0	β2.0
1	100	0	100	0	1.0	1.0	1.0	1.0	1.0
2	0	100	0	100	0.0	0.0	0.0	0.0	0.0
3	90	10	10	90	0.5	0.1	0.278	0.167	0.119
4	90	10	50	50	0.833	0.5	0.735	0.625	0.543
5	90	10	90	10	0.9	0.9	0.9	0.9	0.9
6	50	50	90	10	0.643	0.9	0.682	0.75	0.833
7	10	90	90	10	0.5	0.9	0.549	0.643	0.776

Table 4.1: Evaluation Invariants as Binary Classifiers

Another point worth mentioning is that the *Checker* component can work independently to do the comparisons between different data sets. For example,

we can collect two data sets in different environments, and use one data set to generate invariants, and then check them against the other data set, so that we can find the broken invariants as the difference between the system behaviors under the two environments. We will see this application of this component in Section 5.2.

4.3 Monitor Synthesizer

The last step of our approach is the monitor synthesis. Given a set of generated invariants on messages, architectures, and events, the synthesis process consists of the creation of a node that monitors certain messages, other variables, or events and checks whether they violate any of the invariants at particular program points.

4.3.1 Monitor

We have three sources that need to be monitored: messages on topics, messages on services, and parameters and architecture. For messages on topics, the monitor could simply subscribe to the desired topics, and every time it receives a message it first computes some additional variables (e.g. message frequency, variable variance and change rate), then it checks the corresponding set of invariants, and finally reports if any of them are violated by the message. Since these invariants are simply boolean expressions, such checks are quite straightforward. For messages on a service, the monitor does the same thing, but the difference is that the monitor has to intercept the service communication because of its particular mechanism in ROS. For parameters, the monitor needs to query the ROS master every specified interval, and does the same check as it does to other invariants.

The architecture invariants are expressed as the maximum and minimum set

of nodes corresponding to the jobs such as publishing to a topic, subscribing to a topic, and providing a service. The monitor will query the master node every specified interval to get the architecture information, and then check if any nodes are doing the jobs out of the maximum set boundaries, and if the jobs are being done by the nodes as the minimum set defines. The check interval can be specified through the configuration file as well.

For temporal invariants, we have two kinds of invariants on events publishing to topics and calling services. The monitor has already captured these two kinds of events in the state invariants monitor, so we only need to add the temporal invariant checking when an event of interest happens. For each order-paired interval invariant, the monitor keeps a special state machine with a timer and an events recorder built-in. The special state machine works as shown in Algorithm 11: when the first event of the state machine happens it will turn on a timer and an event recorder; then it records the events other than the second event, and checks the maximum event set; as the same time, if the time-up event happens, it reports the error; and finally when the desired second event happens, it checks the minimum event set and minimum interval time. The time complexity of each check is $O(k)$, where k is the number of ordered-pair interval Invariants that the event is involved.

For user defined pattern analysis invariants, the monitor first initializes the state machines corresponding to the pattern invariants inferred, and then it runs these state machines on each event, and reports the violation if any state machine dies. These machines work as regular state machines, except they have a set of current states inside. Each machine initializes its current states set as all its states. On an event, it drives each state in the current state set to the next state, and removes it when it goes into the error state. The machine dies when current state

Algorithm 11 Check Oredered-Pair Interval Invariant (*event*)

```
1: for interval in interval_invs do
2:   if interval.isClose then
3:     if event is interval.first then
4:       interval.isClose ← false
5:       turn on timer with interval.max_time
6:     end if
7:   else
8:     if event is not interval.second then
9:       add event into interval.current_events
10:      check interval.max_events
11:      report errors if any
12:    else
13:      check interval.min_time and interval.min_events
14:      report errors if any
15:      clear interval.current_events
16:      turn off timer
17:      interval.isClose ← true
18:    end if
19:  end if
20: end for
```

set become empty. The reason is that the monitor may start monitoring at any state of the state machine of the pattern. The algorithm is shown in Algorithm 12, the time complexity of each check is $O(m * k)$, where m is the number of state machines that the event is involved and k is the maximum number of states of the machines.

Algorithm 12 Check Pattern Invariant (*event*)

```
1: for machine in state_machines do
2:   for state in machine.current do
3:     state ← state[event]
4:   end for
5:   remove error states in machine.current
6:   if machine.current is empty then
7:     report violation
8:   end if
9: end for
```

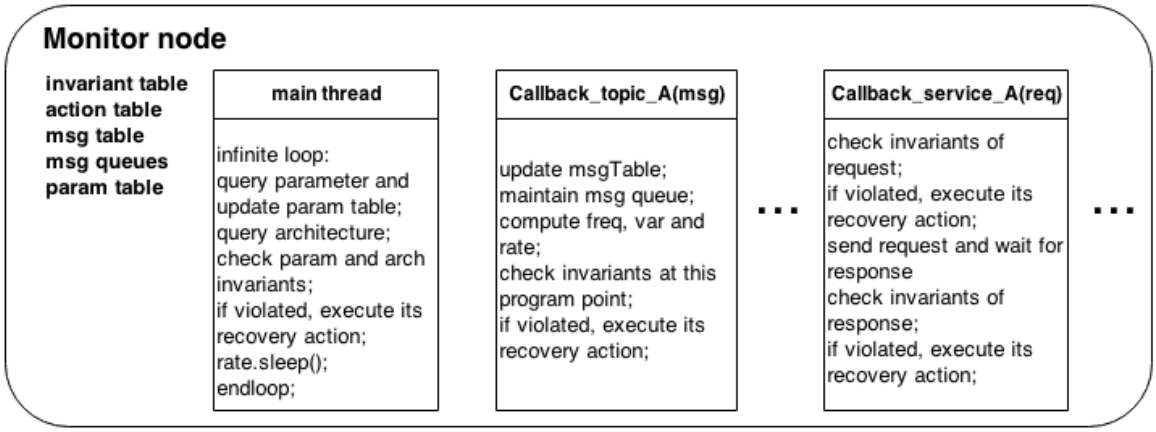

Figure 4.3: Monitor Node Skeleton

The monitor node consists of multiple threads as shown in Figure 4.3. The main thread will query the master node to check the architecture invariant and update the parameter table. When a message is published to a monitored topic, its callback function updates the message table, checks the invariants at this program point, and takes corresponding recovery action if an invariant is violated. When a request is issued for a monitored service, its callback function first checks its invariants on the request message. If an invariant is violated, it takes recovery action and returns false to the client. Otherwise, it sends the request to the real server, and waits for the response. Then, it checks the invariants on response message, and if any one is violated, it takes recovery action and returns false to the client. Otherwise, it returns the response to the client. The number of callback function depends on the topics and services monitored. We implement a central monitor to monitor all the invariants, but it can also be implemented into multiple monitors for different topics or services to better scale up.

4.3.2 Recovery Actions

The monitor also encodes what actions will be taken if any invariant is violated. The recovery actions our approach supports are shown in Table 4.2. The default action is raising a warning as mentioned in the previous section. Others include blocking the bad message, publishing a message, calling a service and unregistering unknown node. Some recovery actions need for the user to declare them in the configuration file as shown in Figure 4.2.

Action	Applied Invariants	When to call	Intercept Communication
Raise a warning	any	Default	No
Block Bad Message	Topic and Service	Users Declared	Yes
Publish a Message	any	Users Declared	No
Call a Service	any	Users Declared	No
Unregister Unknown Publisher	Architecture	Default	No

Table 4.2: Supported Recovery Actions

In the *monitor* tag of the configuration file, the block action defined in *block* tag works for all state invariants on messages on topics and services, where the user can define on which topics or services the monitor is going to drop the messages if they violate any invariants. In Figure 4.2, the monitor will block the "bad" messages on the topics */a/cmd_subject_ctrl_state* and */a/task_waypose*. Thus, for those two topics, the messages are not just consumed by the monitor, but also intercepted and only re-published if they do not violate any invariant. The monitor can also block a service. If the request message breaks any invariants, the monitor will not even send out the request to the real server, while if the response message shows an violation, the monitor will only prevent the server from sending back the response message to the client. This kind of actions can be useful in preventing the system from getting into an abnormal state driven by the "bad" messages.

The block actions need to intercept the communication between nodes. Our approach supports it by remapping the names in the launch file of the ROS system. As an example shown in Figure 4.4, the top part shows the original launch file and the graph view of the program, where the two topics */a/task_waypose* and */a/cmd_subject_ctrl_state* will be monitored. Our approach will first find the nodes who subscribe to the monitored topics, which in this case are */a/pid_ctrl* and */a/ctrl_state_machine*. Then, our approach will localize the two nodes in the launch file and add two name mappings which will make these two nodes subscribe to two new topics */m/a/task_waypose* and */m/a/cmd_subject_ctrl_state*. Finally, the monitor node is added in the launch file, and its graph view is shown as the bottom right part of Figure 4.4, where */m/monitor* is plugged into the original communication channels (*/a/task_waypose* to */a/pid_ctrl* and */a/cmd_subject_ctrl_state* to */a/ctrl_state_machine*), and it looks like a filter to check and relay the messages. The monitor node raises a warning when an invariant is violated by publishing the invariant to the topic */m/broken_inv*. Users can also remove an invariant monitoring by publishing the invariant name to the topic */m/clear_inv* at run-time.

The monitor can publish a particular message on some topic or call some service with some arguments. These actions are defined in the *action* tag as shown in the Figure 4.2. These actions can be applied to any kinds of invariant violations by declaring them in the *violate* tag. For example, in Figure 4.2, the action labeled with f will be taken when any invariant is broken on the program points labeled as a, c, e.

For architecture invariants violation, the system can also prevent unknown publishers. This means that when the monitor detects an unknown publisher (by violating the maximum publishers invariant) registered on the ROS master node,

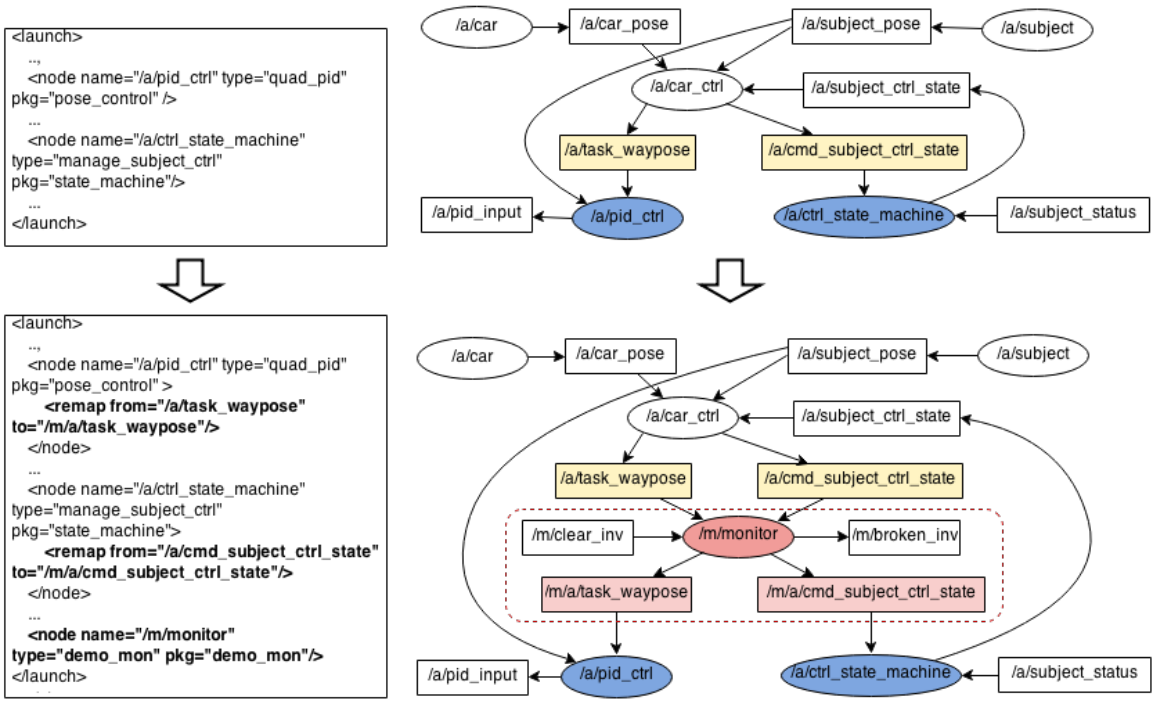

Figure 4.4: Remapping names in ROS programs

it will unregister this publisher to keep the system isolated from this unknown publisher. The intuition is that unknown publishers may have resulted from incorrect remaps or "evil" nodes, which should be prevented immediately for system protection. Subscribers on the other hand do not change the message stream, so their effect on the system is likely smaller. So, the default actions do not prevent unknown subscribers.

Chapter 5

Case Studies

In this chapter, three case studies are presented to assess our approach and explore its potential.

In the first case study, we explore if the invariant monitor with associated actions can reduce failure rate, and how effective are the new invariants at detecting execution anomalies.

In the second case study, we explore another application of invariants, which is the analysis of invariants of different deployments or execution environments.

In the third case study, we focused on investigating temporal invariants.

All the invariant inferring and monitoring tests were conducted on a Mac Pro laptop, which had a 2.5GHz Intel Core i5 processor, a 4GB 1600MHz DDR3 memory and an OS X 10.9.1 operating system.

5.1 Case Study 1: UAV landing on Moving Platform

To start assessing our approach, we applied it on a system designed to land a UAV on a moving platform. The target system was introduced in Figure 1.1 and has

three main components: the UAV (Ascending Technologies Hummingbird [1]), the moving platform (iRobot create [4] with a mounted landing platform of 50cm x 50cm, following its standard "vacuum" motion pattern), and a control system we wrote that tracks the iRobot and directs the UAV in its pursuit. For ease of evaluation, we run the UAV and iRobot in a Vicon [13] motion capture room and provide the UAV with the position of the iRobot.

5.1.1 Training and Evaluation

The training process was conducted under what we determined were normal operating conditions. The UAV can takeoff from anywhere in a 8m x 8m room, the iRobot wanders in the room, and the control system drives the UAV towards the iRobot. The UAV attempts to land on the iRobot when its center is within 15cm of the iRobot's center for 1.5 seconds. These values were driven empirically under normal operating conditions.

To generate invariants for the system, we collected bags from 83 successful runs. We consider a run successful when the UAV lands on the iRobot, turns off its motors, and remains on the platform for 5 seconds. On average, each run took about half a minute.

Among all the messages in the collected bags, we chose those published on four topics containing a total of 56 variables for invariant detection and monitoring. Three topics contained position and attitude information: *iRobot*, *UAV* and *task* (all *double*s). The fourth topic had state information (e.g., startup, launch, hover, task, land, shutdown.) of the controlling system: *state* (unsigned int). As explained in Chapter 3, our tool processes the bag files, clusters the messages around nodes, and packages the traces as required by Daikon. In the end, the trace file for

invariant generation contains over nine million variable-value pairs.

Next, the processed data traces were fed to the extended Daikon inference engine for analysis. Besides the default invariant templates, we activated two of the new invariant templates: time-related and polygon at the time of the assessment we had not implemented the other new invariant templates), and run Daikon twice to get the condition invariants based on the value of *state* messages. The inference process took 6 minutes 20 seconds to generate 1059 invariants from these traces consisting of 465 default, 362 time-related, and 232 polygon invariants. (This process is known to be polynomial with respect to the number of variables [37] so identifying what nodes and topics to monitor, and techniques for reducing the number of invariants to monitor is critical – we further discuss this in Chapter 6). With these invariants and the actions defined in the monitor configuration file, the tool generated the monitor node and a revised launch file so that the monitor can could alongside the original system without the need for recompilation. The recovery actions the monitor encoded are blocking the "bad" messages and publishing a command message to bring the UAV to the *task* state. We did not use the classification/filtering in this case study as that component was developed after this study was conducted.

We evaluated the effectiveness of the invariant monitor on seven different system scenarios (shown in Table 5.1). These scenarios were developed to test the performance of the system with and without the monitor under normal conditions (similar to the training set) and under stress. The stress testing scenarios contain unexpected events that the system developer may not have anticipated, but that the monitor may be able to detect. For the "s3 occupied landing" and the "s7 false airport" scenarios we consider landing as a failure and a canceled landing as a success, while for the other scenarios we set the same criteria for success as set for

the training process.

ID	Name	Description	Success Certiria
$s1$	Normal	Same as training conditions.	Succeeds on landing.
$s2$	Wind Blowing	$8-38$ KPH wind.	Succeeds on landing.
$s3$	Occupied Landing	Platform is occupied by another object.	Succeeds if it avoids landing.
$s4$	Fragile Platform	Platform will tip if the UAV lands near the edges.	Succeeds on landing.
$s5$	Slowed Link	iRobot position information given at a slower rate.	Succeeds on landing.
$s6$	Stealing Vehicle	Fake iRobot position is manipulated to "steal" the vehicle.	Succeeds on landing.
$s7$	False Airport	iRobot position is incorrect and no vehicle is located there.	Succeeds if it avoids landing.

Table 5.1: Evaluation Scenarios.

5.1.2 Results

For each of the scenarios, we performed 5 trials with and without the invariant monitor. Table 5.2 summarizes the results. Over all the test scenarios, the base system without the monitor succeeded 23.8% of the time, while with the monitor it succeeded 89.4% of the time. Figure 5.1 plots the success rates for each scenario. The system with the monitor worked more safely that it did without the monitor, as it succeeded with a higher rate for all the scenarios.

For the successes, the base system took an average of 35.5 seconds to succeed, while the system with the monitor took 62.8 seconds to succeed. Figure 5.2 shows a box plot depicting the average time in seconds with and without the monitor and the variance in these measurements (only for the scenarios in which the system without the monitor successfully landed). Without the monitor, the average time has a low variance within each scenario and over all scenarios. With the monitor there is a high variance in the time to success. This is because the monitor tends to be conservative, as it only allows the UAV to land when all the invariants are

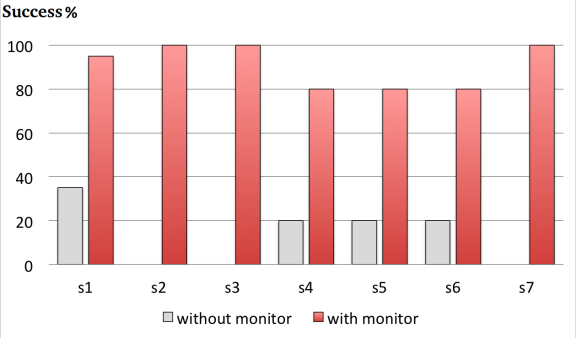

Figure 5.1: Landing success rate

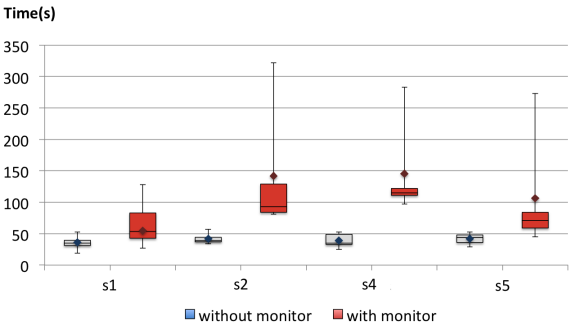

Figure 5.2: Time to land

satisfied. In the best case, this will happen on the first attempted landing, but in most cases it requires a number of attempts. Also, to monitor the invariants the monitor adds, on average, a 0.35ms latency to the published messages.

5.1.3 Detailed Analysis

We now look at the details for each of the scenarios. We first describe the "normal", "wind blowing", and "fragile platform" in more detail since they let us introduce different types of invariants and contexts, and then briefly discuss the other scenarios. A summary of the results per scenario is available in Table 5.2.

In the "Normal" scenario, most failures were caused by the iRobot's suddenly changing direction while the UAV was trying to land. Figure 5.3 shows the

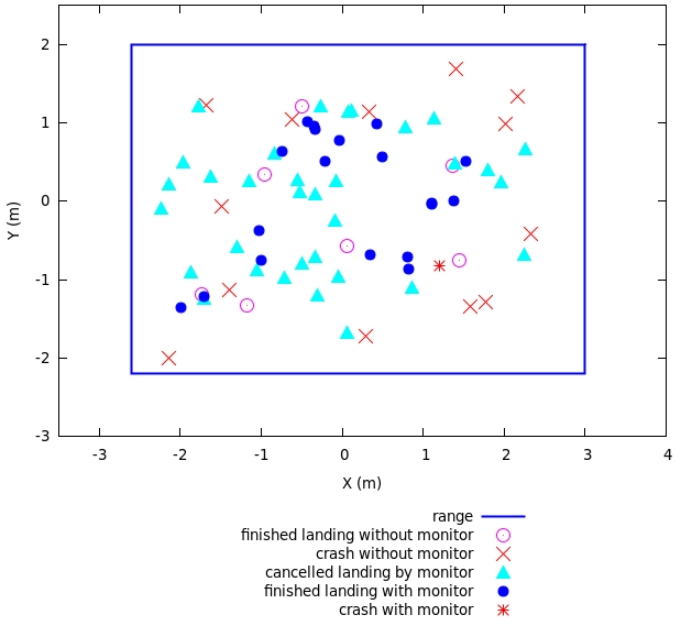

Figure 5.3: Outcomes under normal scenario.

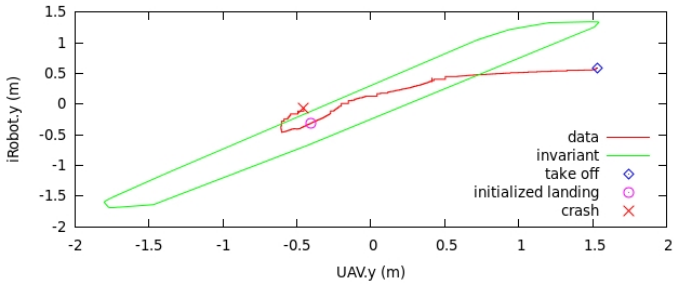

Figure 5.4: Normal scenario without monitor.

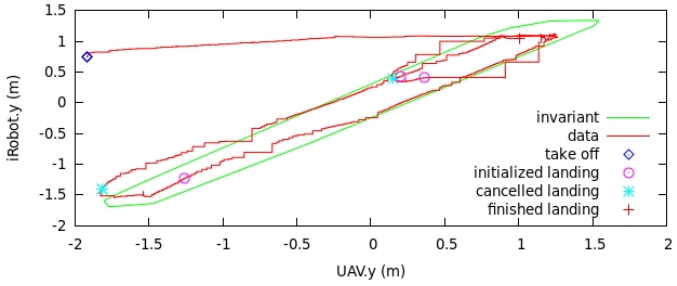

Figure 5.5: Normal scenario with monitor.

	Scenario		% Success	Avg. Time to Land (sec)	Sample Invariants Broken During Failure
Without Monitor Successes 23.8%	$S1$	Normal	35	35.5	$polygon(UAV.x, iRobot.x)$ $polygon(UAV.y, iRobot.y)$
	$S2$	Wind blowing	0	42.25	$polygon(UAV.x, iRobot.x)$ $polygon(UAV.y, iRobot.y)$ $polygon(IMU.roll, IMU.acc\_y)$ $polygon(IMU.nick, IMU.acc\_x)$
	$S3$	Occupied landing	0	-	$UAV.z \leq 0.371295$
	$S4$	Fragile platform	20	39	$-0.0593147 \leq UAV.rx \leq 0.145754$ $-0.106682 \leq UAV.ry \leq 0.0836237$
	$S5$	Slowed Link	20	42	$freq(iRobot) \geq 2.04876$
	$S6$	Steal vehicle	20	41.6	$-0.457771 \leq rate(iRobot.x) \leq 1.01126$ $-0.532218 \leq rate(iRobot.y) \leq 0.962376$
	$S7$	False airport	0	-	$UAV.z \geq 0.245868$
	Scenario		% Success	Avg. Time to Land (sec)	Reinitiated Landings
With Monitor Successes 89.4%	$S1$	Normal	95	62.8	1.7
	$S2$	Wind blowing	100	141.8	4.8
	$S3$	Occupied landing	100	-	-
	$S4$	Fragile platform	80	145.6	16.2
	$S5$	Slowed Link	80	106.4	6
	$S6$	Steal vehicle	80	147.6	-
	$S7$	False airport	100	-	-

Table 5.2: Summary of results across all scenarios.

successful and failed landings with and without the monitor in the test area where the iRobot was operating. The thicker rectangle indicates the boundary of the area. The iRobot will typically drastically change directions when it hits a wall, although it occasionally chooses to follow the wall. That is why most of the crashes without the monitor are located towards the borders. The single failure with the monitor occurred as the UAV landed on the platform but slid off of it because of its incoming speed (even though the speed was within the limits of training scenarios). When the iRobot quickly changes direction, the monitor detects violations of one of the inferred polygon invariants which characterize the relations between the UAV

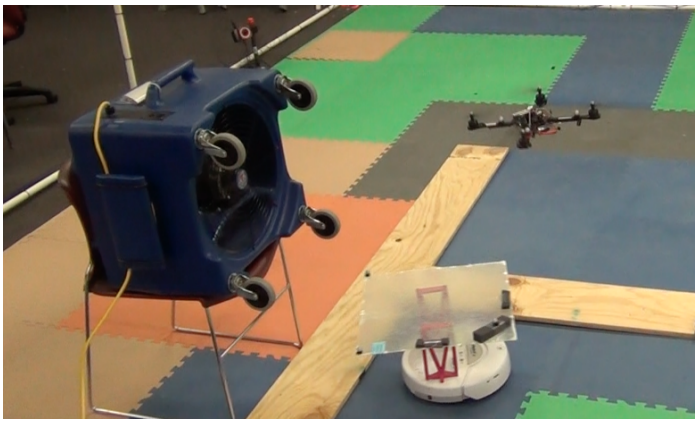

Figure 5.6: Wind Blowing Scenario

and *iRobot* positions, speeds, and rotations during the landing process. Figure 5.4 shows the y axis polygon invariant between the UAV and iRobot ($UAV.y + 0.0554 * iRobot.y \geq -1.89 \cap UAV.y - 0.990 * iRobot.y \leq 0.151 \cap UAV.y - 1.081 * iRobot.y \geq -0.202 \cap UAV.y + 1.732 * iRobot.y \geq -4.664 \cap ...$) without the monitor running. When the UAV takes off, it is outside of this constraint. It then moves over the iRobot and initiates the landing sequence. As seen in the figure, the UAV violates the polygon invariant while still trying to land and crashes. In contrast, Figure 5.5 shows the same scenario with the monitor enabled. In this case, whenever the invariants are violated, the landing is restarted. Eventually, the UAV is able to successfully land while staying within these constraints.

In the "wind blowing" scenario (see Figure 5.6), the strong wind breaks many invariants derived from the normal setup. Neither the system, nor the monitor were designed to explicitly consider wind. However, the monitor is able to detect violations of the UAV and iRobot positions and the roll and acceleration of the vehicle, as described in Table 5.2. Figure 5.7 shows the locations where landings occurred. None of the landings occurred within 2 meters of the blower where the wind speed was up to of 33 KPH, which prevented the landing sequence. Even

away from the fan, the system without the monitor was unable to successfully land. The system with the monitor was able to detect constraint violations to prevent the landing when it was unsafe and was able to land every time. Figures 5.8 and Figure 5.9 show two of the trials with and without the monitor for the polygon invariant involving the UAV pitch and acceleration on the x-axis. In Figure 5.8 the UAV leaves the polygon and crashes almost immediately. In Figure 5.9, however, the violation of the invariant while using the monitor leads to a landing reinitialization, avoiding a crash (other monitored invariants were violated within the polygon leading to other landing reinitialization as well).

In the "fragile platform" scenario (see Figure 5.10), the landing platform would tilt if the UAV did not land in the upper right quadrant as shown in Figure 5.10. The monitor detected the error when checking the violation of the invariants on iRobot.rx and iRobot.ry which indicate the horizontal angle of the platform. Figure 5.12 shows one of the angles without the monitor. The straight lines indicate the bounding constraint inferred. As shown by the line, the UAV started to land on the platform, but then the platform tilted and the UAV fell off and crashed. Figure 5.13 shows the same setup with the monitor. In this case, the UAV initialized landings three times, but in the first two the landing was canceled when the constraints were violated. Overall, with the monitor the UAV was able to successfully land 80% of the time, while without the monitor it was only successful 20% of the time.

In the "occupied landing" scenario, the monitor detected that the platform was occupied since it could not decrease its height to match that of the platform as it did in the normal case. The invariant is shown in Table 5.2 as $UAV.z \leq 0.371295$, which means in normal case the height of the UAV should be lower than 0.371295 to finish its landing sequence. The monitor detected the violation on the invariant,

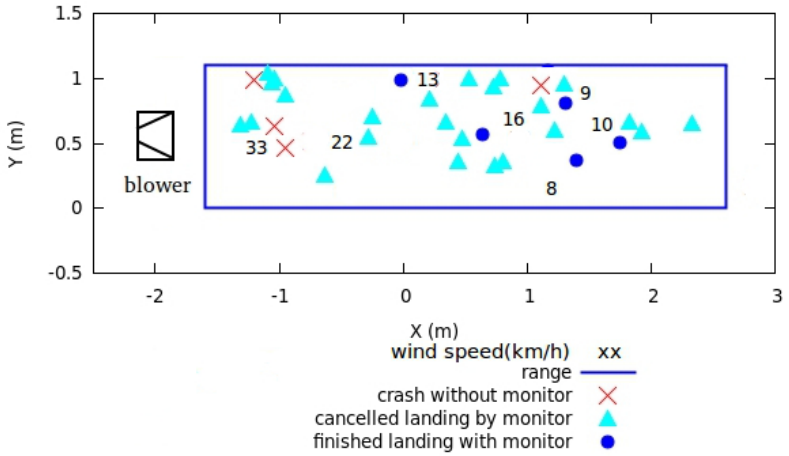

Figure 5.7: Outcome under wind blowing scenario.

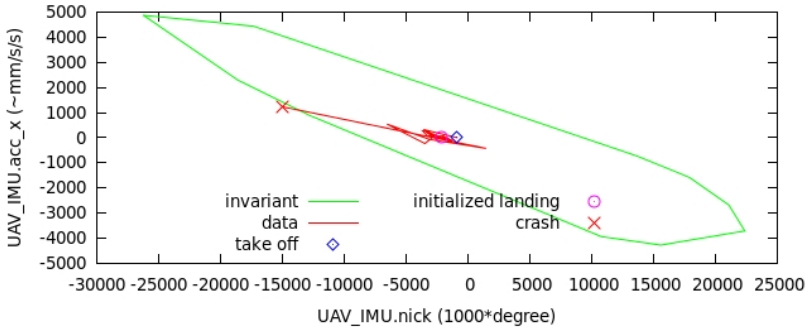

Figure 5.8: Wind blowing scenario without monitor.

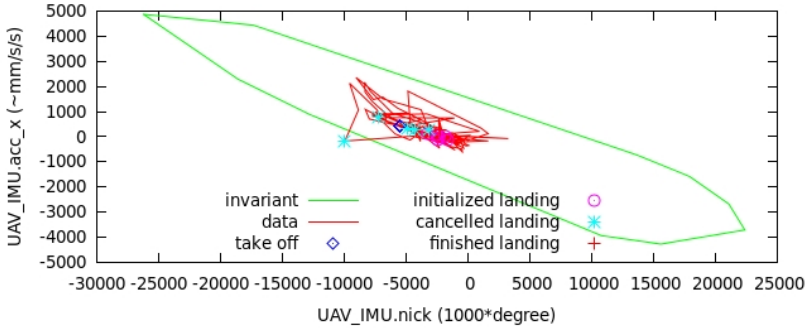

Figure 5.9: Wind blowing scenario with monitor.

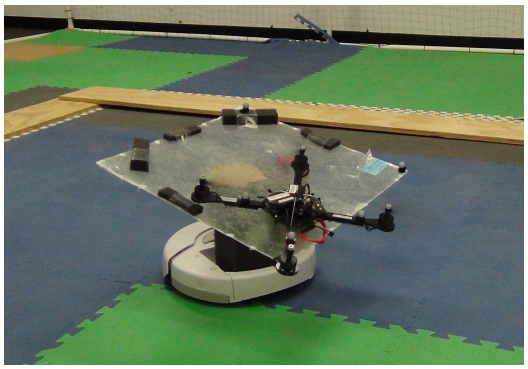

Figure 5.10: UAV attempts to land on fragile platform.

and it canceled the landing, which we consider a success.

In the "slow link" scenario, the message rate from the iRobot position was periodically (every 5 seconds) slowed down to 0.5Hz. to mimic a faulty positioning sensor or a radio link that drops packets. The monitor detected this abnormal situation by the invariant $freq(iRobot) \geq 2.04876$ on message frequencies as shown in Table 5.2. When the position of the iRobot was published at a low frequency, the monitor interrupted the landing sequence to avoid crashes as it thought the link was not reliable. And it only allowed the UAV to land at normal publishing frequency.

In the "stealing vehicle" scenario, we published fake iRobot positions to try to get the vehicle to land in another location when the iRobot was moving in the upper half of the target area. The monitor detected this anomaly through a violation of the invariant on the change rates of position messages. In Table 5.2, the invariant are $-0.457771 \leq rate(iRobot.x) \leq 1.01126$ and $-0.532218 \leq rate(iRobot.y) \leq 0.962376$, which indicated the ranges of the iRobot's speed. In this case, when the positions of the iRobot changed too quickly, the UAV kept flying without landing on either the false or the right platform. When the iRobot was moving in the lower half of the cage and no other location was published, the UAV would try to land,

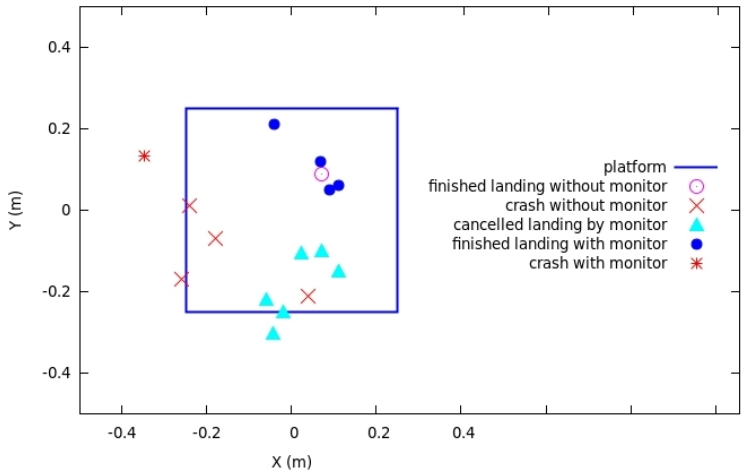

Figure 5.11: Outcome under fragile platform scenario.

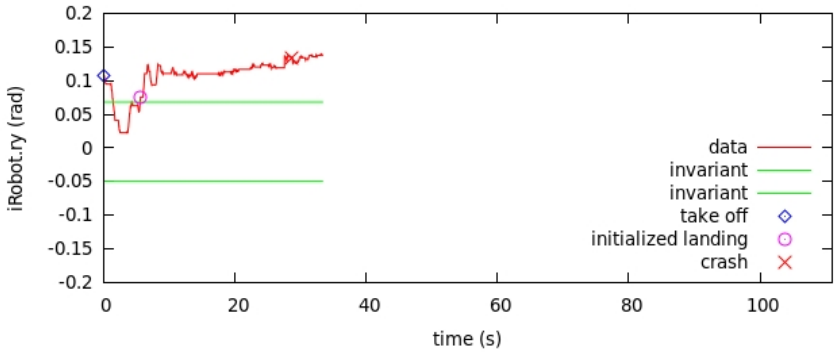

Figure 5.12: Fragile platform scenario without monitor.

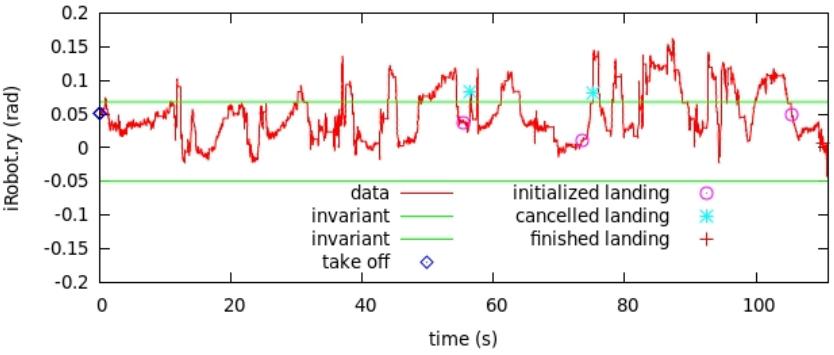

Figure 5.13: Fragile platform scenario with monitor.

which we considered a success.

In the "false airport" scenario there was no iRobot, but rather a false location was published. If the false location was outside the region where the UAV had previously seen the iRobot, then the UAV refused to go to that location and filtered out these false messages. If the false location was in the correct range, the UAV attempted to land. However, the monitor could tell the difference of the height between the false and correct platforms, so the UAV with the monitor would not land on the false airport.

In sum, in this case study we can see:

- The inferred invariants monitor increases the system success rate when faced with unexpected situations although its efficiency may suffer;
- Although existing invariant templates serve to detect execution anomalies, the two new invariant types (2-D polygon and time-related) contributed to the detection of execution anomalies. In four out of the seven scenarios, the anomalies can only be detected by the new invariants.

5.2 Case Study 2: Water Sampling

In this case study, we want to explore the usage of this tool on a system that we did not implement and that is subjected to environmental changes. Since invariants are known to be useful in analyzing a program's evolution, we conjecture that, in robotics, invariants may also be helpful in detecting problematic evolution of the environment.

We conducted this analysis on the bag files collected from indoor and outdoor flight tests of the Water Sampling system [30], which is designed to automatically

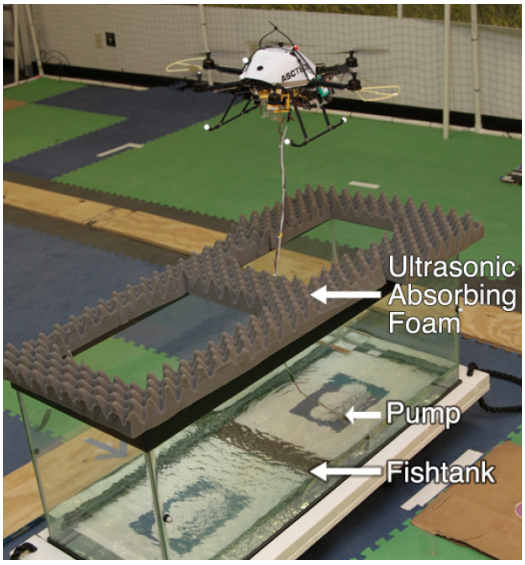

Figure 5.14: Indoor Water Sampling

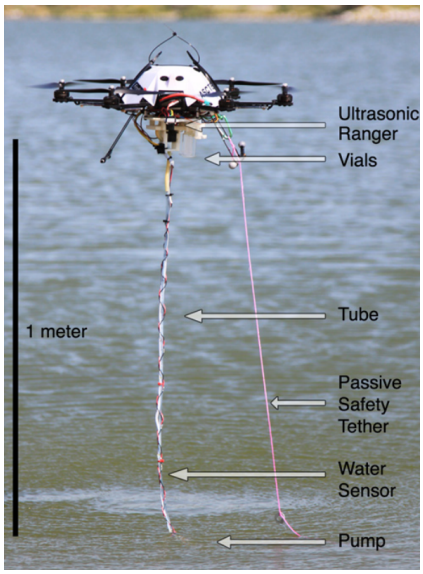

Figure 5.15: Outdoor Water Sampling

```
−<imROS project="water_sampling">
 −<scope>
    <publish id="a" conds="1">/a/water_sampler_board_raw</publish>
    <publish id="b" conds="1">/a/subject_pose</publish>
    <publish id="c" conds="1">/a/robot_imu</publish>
    <arch id="d"> pub,sub </arch>
    <temporal id="e" events="a,b,c" conds="1"/>
   −<condition label="1">
      <topic>/a/pump_control</topic>
      <value>pump_control==1</value>
    </condition>
  </scope>
 −<detect inv="water_sampling.inv">
    <bag> bags/indoor </bag>
  </detect>
 −<check inv="water_sampling.inv">
    <bag> bags/outdoor </bag>
  </check>
</imROS>
```

Figure 5.16: Configuration for Water Sampling System

fly over a body of water, approach particular locations, and sample the water through a pump, as shown in Figure 5.15. Since the pump powered by the UAV's battery can only work within about 1 meter height, the most challenging part of this system is the height control. To avoid crashing into the water while collecting samples, the height control needs to be precise, thus the system uses a combination of ultrasonic sensor, GPS, and conductivity sensors to estimate the relative height.

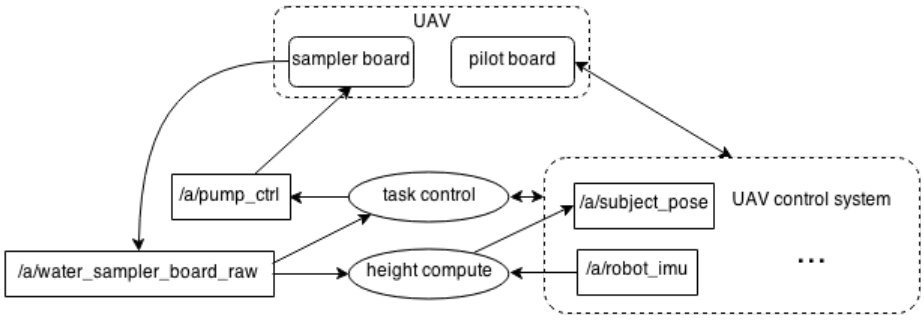

Figure 5.17: Water Sampling System

5.2.1 Training

The system flow is shown in Figure 5.17, which extends the UAV system used in the first case study with some new components. A sampler board is added on the UAV to control the water pump and report sensors' data. The sensors include two ultrasonic sensors and five conductivity sensors. The ultrasonic sensors indicate the relative height of the UAV, and the conductivity sensors tell whether the pump is actually in the water. These data are sent to the topic */a/water_samler_board_raw* in the remote control system, where two components *heightcompute* and *taskcontrol* are added. The *heightcompute* component first gets the height from ultrasonic data, and combines the height from */a/robot_imu* to compute the more precise relative height, and then updates the */a/subject_pose* message. Based on the UAV status, pump status and other information, the *taskcontrol* component controls the task flow including how to fly the UAV and when to turn on/off the pump.

The system was first tested in a controlled environment, as shown in Figure 5.14, where the UAV flew over and sampled water from a fish tank protected by a ultrasonic absorbing foam. We collected bags from 16 successful indoor runs, when the UAV started 2-3 meters away from the fish tank, flew over the tank, then descended and sampled water three times, and finally flew back and landed.

Having talked to the developer, we chose three critical topics to monitor, which are */a/water_sampler_board_raw*, */a/robot_imu* and */a/subject_pose* as shown in Figure 5.16. Those topics capture new or important information about the system including the simpler board data and the UAV attitude data. We also focused on the messages when the system was in the water sampling state (when the *pump_control* was 1). We instantiated the default invariant templates, and our new time-related, polygon, architecture and temporal invariant templates. We generated a 49.2MB data trace with 44 variables, and inferred 711 invariants including 242 ones from Daikon's default templates, 77 time-related, 385 polygon, 4 architecture and 3 temporal ones from our extended invariant templates in 17 seconds.

We first evaluated if the bags were sufficient to represent the program behaviors in the indoor environment by checking the inferred invariant set's stability. We conducted it by picking one bag out of the training set, and using this bag to check the invariants generated from the remaining 15 bags. If the invariants were not violated by the checking bag, we considered the invariants stable. We randomly picked 3 bags as the checking bags (one at a time) to do the check, and the result is shown in Table 5.3.

We observed that the default, time-related, architecture and temporal invariants were stable, but 35% of the polygon invariants were violated in the third check. The reason is that the polygon invariants are built from combinations of variables at one program point, no matter whether they are dependent or not. For example, our approach will infer a polygon invariant combining the UAV's speed and the UAV's heading angle. They are independent variables, but with a limited set of runs, the data may not show the maximum and minimum speed when the UAV is heading in different directions. So this type of invariants needs more runs to support it. Therefore, we used the 13 indoor runs to generate the invariants, and

then use the other 3 indoor runs to classify and filter out fragile invariants. Finally, we got 533 invariants, which includes 229 default, 64 time-related, 4 architecture, 3 temporal and 233 polygon invariants.

Stability check	Default invs/violated	Time-related invs/violated	Polygon invs/violated	Architecture invs/violated	Temporal invs/violated
1	242/2	77/4	385/69	4/0	3/0
2	242/2	77/0	385/24	4/0	3/0
3	242/6	77/3	385/135	4/0	3/0
retained	229	64	233	4	3

Table 5.3: Stability Check Result

5.2.2 Evaluation

We collected a bag from one outdoor run, which successful flow about 30 meters over the lake of $1.091(km)^2$ and sampled water from it. We first checked the default, time-related and architecture invariants invariants inferred indoors against this outdoor bag, where we found that 24 invariants were broken, while there are 276 unbroken invariants shared by the indoor and outdoor environments. The broken invariants indicate environmental changes from indoor to outdoor projecting to the UAV's behaviors, as shown in Table 5.4.

The frequency invariant tells the change of navigation signals frequencies, where the frequency of outdoor GPS is slower than the VICON system we used indoor. The broken invariants about *pitch* and *roll* indicate the difference of the UAV's attitude, which could be caused by the winds outdoor. The violation of the invariants about the acceleration is also a direct result of the more windy environment outdoor or more aggressive maneuvers. The architecture invariants are broken because the system switches from the indoor VICON system to the outdoor GPS navigation, which used different nodes. The one violated ordered-

pair temporal invariant was caused by the low-frequency GPS signal which made the interval greater than $0.0621s$.

Invariant Checked	Invariant Violated	Details
300	24	$sampler\_raw.H2O_1 >= 306$ $Var(sampler\_raw.H2O_5) <= 14.0$
		$pose.rotation.x <= 0.0535418$ $pose.rotation.y >= -0.0804302$ $pose.translation.x >= -1.99948$ $pose.translation.y < pose.translation.z$ $pose.translation.y <= 0.32948$ $Freq(pose) >= 20.0$
		$imu.acc\_angle\_nick >= -4284$ $imu.acc\_angle\_roll < imu.mag\_z$ $imu.acc\_angle\_roll <= 2516$ $imu.acc\_x\_calib >= -747$ $imu.acc\_y\_calib >= -439$ $imu.angle\_nick >= -5099$ $imu.angle\_roll <= 3192$ $imu.angvel\_nick < imu.mag\_z$ $imu.angvel\_nick <= 2173$ $imu.height\_reference! = 0$ $imu.mag\_x <= 772$ $Var(imu.angle\_yaw) >= 5934.0$
		$MaxPubs(pose) = [vicon]$ $MinPubs(pose) = [vicon]$ $MaxSubs(gps) = []$
		$imu \rightarrow pose : \{0.0000524, 0.0621, \phi, 2 \times raw + 2 \times imu\}$

Table 5.4: Check Result of Outdoor Testcase

From the broken invariants, we can learn how to improve the indoor testing of the system to better mimic the outdoor environment. For example, we could tune the indoor navigation signal frequency to match the outdoor one, or use a fan to simulate the windy environment. These enriched indoor tests could reduce the risk when deploying the UAV outdoor.

For polygon invariants, the result showed that 174 out of 233 invariants were violated as well. All the polygon invariants related to the variables associated with the violated invariants in Table 5.4 were also violated. For example, the polygon of

imu.acc_angle_nick and *imu.mag_y* were violated, though the two variables were not quite related to each other. Given the number of violations, we conjecture that polygon invariants need further pruning to be useful.

5.2.3 Checking User Assumption

Through our interactions with the developer of the system in the training process, we were able to find some unstated and wrong assumptions. For example, the developer assumed there should be an invariant indicating that, when the pump is on, the sensor should always be wet as shown in conditional invariant $pump\_on = 1 \Rightarrow sampler\_raw.wet >= 670$. However, our inference system did not find such invariant so we proceeded to investigate why. To pinpoint the potential bad data, we set the invariant list with this supposedly missing invariant, and then used the generated checker to check all the bags. The checker found the data that violated this invariant. We provided this finding to the developer, and ended up finding the problematic code in the on-board pump controller that caused the absence of this invariant. We found that, if the sensor was not wet in the sampling state, the pump controller would still try to turn on the pump every 0.4 seconds, and fail immediately. However, it would set the pump state to be 1(on) in the message, even if it failed to turn on the pump. That is why we found many such pulses on the pump state when the sensor was not wet. In addition, while we were looking into the data, we found another strange behavior with pulses of the pump state when the sensor indicated wet. The pattern looked very similar when the sensor was not wet. The developer confirmed the problem. While a node published the message indicating that the sensor was wet, the on-board controller felt the sensor was not wet, because they were using different thresholds to determine wet or dry.

In sum, this case study shows two other applications of our tool:

- It can be used to check user expectations and assumptions, and pinpoint the context of the inconsistencies if there is any violation.
- It can be used to check differences of system behaviors under different environments or deployments.

5.3 Case Study 3: Crop Surveying

In the third case study, we worked with a system that utilized a UAV and a small laser scanner to measure crop heights as shown in Figure 5.18. It processes the cluttered laser reflection data in real-time to determine both the distance to the ground and to the top of the crops to allow users to precisely control the height of the UAV. From the view of the UAV's control, it provided more precise relative height data from a laser sanner when flying over the crop. And it applied a Kalmen filter to filter out some clutters in the raw scanner data. Intuitively, the integration of the new height computation component needs to be the focus of testing, since we need to know how well it integrates with original system. In this case study, we also explore the application of temporal invariants.

5.3.1 Training

This system is still based on the structure of the UAV control system we used in the previous case studies, but it remaps topics and adds three nodes as shown in Figure 5.19, where each ellipse represents a node and each rectangle a topic. The *scan* node generates the laser scan messages, and the *imu_laser_sync* node synchronizes and pairs the *scan* messages and *imu* messages. The *kalman_height*

Figure 5.18: Crop Surveying

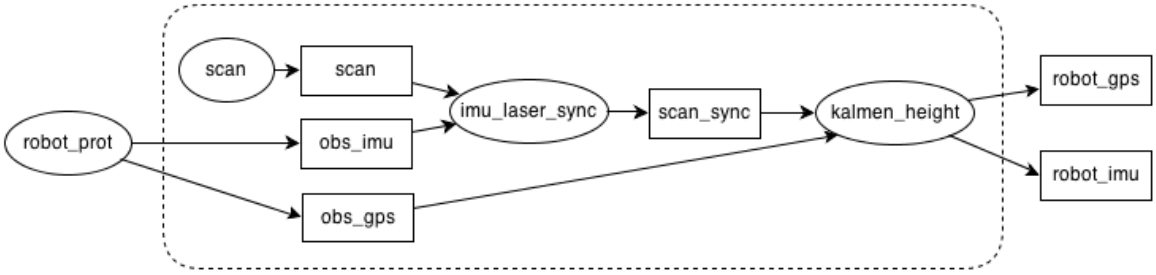

Figure 5.19: New Components

node computes the height based on pairs of *scan* and *imu* messages. Thus, these nodes and topics compose a subsystem (as shown in the dashed rectangle in Figure 5.19), which computes the more precise height and updates the field in the original messages.

From the structure of the system, it makes sense to monitor the six related topics. The subsystem has some synchronized behaviors, as the node *imu_laser_sync* has to consume two messages from the two topics *scan* and *obs_imu* individually, and then generates a new message on the topic *scan_sync*. We inferred temporal invariants and architecture invariants as shown in the configuration file 5.20. The developer provided 13 bags from his tests, which were conducted at the lab when the UAV flew over fake crop as illustrated in Figure 5.21 successfully. Because the system

```
−<imROS project="crop_surveying">
 −<scope>
    <publish id="a"> /dave/scan </publish>
    <publish id="b"> /dave/obs_imu </publish>
    <publish id="c"> /dave/scan_sync </publish>
    <publish id="d"> /dave/obs_gps </publish>
    <publish id="e"> /dave/robot_gps </publish>
    <publish id="f"> /dave/robot_imu </publish>
    <arch id="g"> pub,sub </arch>
   −<temporal id="h" events="a,b,c">
      <pattern> ((A|B)+ C)* </pattern>
    </temporal>
  </scope>
 −<detect inv="crop.inv">
    <bag> bags/crop </bag>
  </detect>
 −<monitor launch="demo.launch" inv="checked.inv">
   −<action id="z">
      <topic> /dave/robot_gps </topic>
      <value> /dave/obs_gps </value>
    </action>
   −<action id="y">
      <topic> /dave/robot_imu </topic>
      <value> /dave/obs_imu </value>
    </action>
    <violation> g -> z,y </violation>
  </monitor>
</imROS>
```

Figure 5.20: Configuration for Crop Surveying System

Figure 5.21: Fake Crops

was still under development, we only used those bags for invariant inference without classification, filtering and monitoring. The process generated a 116.3MB data trace, and inferred 783 invariants including 293 default, 133 time-related, 350 polygon, 4 architecture and 3 temporal in 46 seconds.

5.3.2 Evaluation

We found an interesting invariant, an instance of an extension we performed on an array variable *ranges*: $0.12984 <= std(scan.ranges) <= 1.4126$. This variable is from *scan* message, which contains an array representing the ranges detected by the laser scanner on different directions. The invariant captures that the standard deviation of the *ranges* in one scan should be greater than or equal to 0.12984. This makes sense because when the UAV is flying over the crop, the laser scanner should always give varied values in one scan. If the ranges all have the same values, there is probably something wrong with the scanner. For example, if the scanner is blocked by an object or all the objects are out of the scanner's maximum range, we will get the same values. So this invariant can detect that error and report it users. On the other hand, we can imagine that when the UAV is going to land on a level ground, the standard deviation of the *ranges* should be small. We could get an invariant in the landing state like $std(scan.ranges) <= 0.3$. If we are trying to land on a uneven ground or crops, this invariant will be violated and raise warnings.

We were also interested in temporal invariants, because we knew there was a synchronization node *scan_sync*, which paired the closest two messages on the topics of *scan* and *obs_imu* into a new message on the topic *scan_sync*. The paired messages should be close enough to make the *kalman_height* node generate a precise enough estimate. The inferred ordered-pair interval invariants indicated this property as shown in Table 5.5. The first column shows the interval between the *scan* messages and the *obs_imu* messages, which were highly interleaved. The maximum interval between these two messages was 1.139*s*. If the two messages arrive with quite different time stamps, the *scan_sync* node may still pair them and

put it to the *kalman_height* node which may generate the wrong height estimate. However, the monitor would detect that abnormal behavior and prevent the error's propagation.

We also tried to infer the pattern invariants $((A|B)^{+}C)^{*}$ on the events of publishing to topics *scan*, *obs_imu* and *scan_sync* as shown in the configuration file in Figure 5.20. We expected there would be $((scan|obs\_imu)^{+}scan\_sync)^{*}$ and $((obs\_imu|scan)^{+}scan\_sync)^{*}$, because the node *imu_laser_sync* had to receive the at least one message to publish the new synchronized message. But there was no such pattern in the bag's event sequence. We see in Table 5.5 that there are consecutive *scan_sync* messages, which break this pattern. The precise pattern may not show up in the event sequence because of the unsynchronized message subscription mechanism of ROS system, which means messages publishing order may change in messages receiving order. In this case, there might be the case that the *imu_laser_sync* node first receives and processes a *scan* message and then publishes a *scan_sync* message, but the recording node may received the *scan_sync* message before the *scan* message. So the patterns $((scan|obs\_imu)^{+}scan\_sync)^{*}$ or $((obs\_imu|scan)^{+}scan\_sync)^{*}$ are not appropriate.

As we discussed it with the developer, we realized that this pattern did exist in the view of the node *imu_laser_sync*. However, because of the high frequency of the messages (10Hz) and the unsynchronized message subscription mechanism of ROS, the recording node received a different sequence of messages, which violated the pattern.

The last thing we want to discuss are the architecture invariants. As we can see in Figure 5.19, the laser scanner subsystem sits between the *robot_prot* node and the *robot_imu* and *robot_gps* topics. If anything happens in the subsystem that breaks the connection, the *obs_gps* and *obs_imu* messages cannot pass through,

Interval	**scan → obs_imu**	**obs_imu → scan**	**scan_sync → scan**
Min Interval(s)	0.103	0.10	0.10
Max Interval(s)	0.995	1.139	1.31
Min Events	ϕ	ϕ	ϕ
Max Events	$5 * scan$ $4 * scan\_sync$	$4 * obs\_imu$ $4 * scan\_sync$	$3 * obs\_imu$ $3 * scan\_sync$

Table 5.5: Interval Invariants

which may cause the UAV's crash. For example, the messages would be blocked, if the node *kalmen_height* dies. By monitoring the architecture invariants, this failure can be detected by the violations of the minimum publishers and subscribers invariants as shown at rows 3, 4, 5, 6 in Table 5.6. The recovery action defined in the configuration file 5.20 can be taken, where the monitor will relay the messages to provide the position and attitude information continuously.

ID	**Topic**	**max pubs**	**min pubs**	**max subs**	**min subs**
1	*scan*	*scan*	*scan*	*imu_laser_sync*	*imu_laser_sync*
2	*obs_imu*	*robot_prot*	*robot_prot*	*imu_laser_sync*	*imu_laser_sync*
3	*obs_gps*	*robot_prot*	*robot_prot*	*kalmen_height*	*kalmen_height*
4	*scan_sync*	*imu_laser_sync*	*imu_laser_sync*	*kalmen_height*	*kalmen_height*
5	*robot_gps*	*kalmen_height*	*kalmen_height*	*robot_trans*, *robot_monitor*	*robot_trans*, *robot_monitor*
6	*robot_imu*	*kalmen_height*	*kalmen_height*	*robot_trans*	*robot_trans*

Table 5.6: Architecture Invariants

In this case study we found that:

- The interval invariant can constrain the system's temporal behaviors at some level, but because of the unsynchronized message passing system, precise temporal patterns may be hard to detect;
- The architecture invariants can be quite effective, and may be very useful to pair the architecture reconstructing actions, like repairing the broken channels by relaying messages.

5.4 Limitations

The case studies illustrated the potential of the approach, but it also put in evidence its current limitations. One kind of limitation is when abnormal behavior cannot be detected by the invariants. A second kind of limitation is when the invariant is violated but it does not lead to a failure.

One reason of the first kind of limitation is the invariant's approximation of the system's correct behaviors. For example, with a value trace of variable v such as $\{1, 5, 2, 4, 2, 5\}$, we infer an invariant $1 \leq v \leq 5$. This invariant puts a constraint on the variable in terms of the range according to the data trace, but it also relaxes the constraint from the data trace, since it allows the variable to be 3, which did not happen in the data trace. The same thing happens with polygon invariants, which can detect a 2-D space boundary by a convex hull; however, it cannot capture the space boundary of an non-convex hull. Also, the polygon invariant cannot detect a dangerous hole in the space which may crash the UAV either. This kind of limitation is rooted from the fact that the inferred invariants are approximations of system's correct behaviors. Actually, we cannot expect an analysis tool without such limitation, otherwise this analysis tool is an even better implementation than the original system. As long as an analysis technique can detect some kind of bugs with reasonable trade-off compared to the gain, the technique has the potential to be valuable.

Another reason for the first kind of limitation is the model approximation, as our approach only uses some signals (messages, architectures and parameters) to represent the whole system state, which are not enough to detect all the anomalies. For example, the anomalies laid in the variables other than messages cannot be detected by our approach. Some relations among two messages may exist,

but our approach does not consider them because of their long distance on the publish/subscribe chain. In fact, all analysis techniques have to make decisions on the model approximation. Our approach sets the granularity on the message level based on the scale of the robotic system, and in the experiments we can get reasonable gains with this granularity.

Our approach also generates some false positives, when some invariants are violated but they do not lead to a failure. Here, most false positives are caused by insufficient training runs which are used to generate the invariants. Although we provide some justification and classification methods to reduce their effects, they cannot eliminate all false positives. And different types of invariants may need different numbers of training runs to make them stable. As shown in the second case study, polygon invariants need more training runs than 1D-range invariants. In addition, some false positive invariants in one scenario may be useful to detect anomalies in other scenarios. For example, we may use one kind of platform to training the UAV-landing system. Some invariants inferred indicate that the system is angle-sensitive, which are useful to detect some anomalies. However, when we change it to another kind of platform which makes the system not that angle-sensitive, these invariants become the false positives. For those reasons, we made the invariant monitor adjustable at runtime, where users can remove any invariants as they want.

Apart from the overhead in the invariant inference, the monitor also brings in run-time overhead in terms of message-relaying latency. We measured it in the first case study, where to monitor the 1059 invariant the monitor introduced a 0.35ms on average latency in relaying each message. This is an acceptable overhead in our case, but it may become unacceptable when dealing with real-time or larger scale systems. Although we have not tried, having distributed monitors may alleviate

the run-time overhead.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have introduced a general approach for automated invariant inference and monitoring, and implemented it in the context of ROS so that any system implemented with this operating system can leverage it with minimal effort. The approach is able to automatically infer rich invariants for a robotic system based on a training set, and it was able to detect the violation of those invariants and avoid failures under various scenarios. The case studies illustrated the potential of the technique and toolset in error detection and potential recovery when facing unexpected situations.

The approach includes new invariant templates that account for properties that are deemed important in the context of robotic systems, such as time-related, polygon, architecture, and temporal invariants. Invariant classification was also built in the approach, which can help to refine the invariant set to reduce the number of false positives. In the invariant monitor, recovery actions can be defined to be triggered when anomalies are detected.

6.2 Future Work

Besides more extensive empirical assessment of the approach we see several technical avenues for future work.

First, we would like to study how to increase the scalability of the approach. For invariant generation, we are investigating the application of filters based on the variance and pedigree of a variable as well as the automatic identification of redundant messages. Within invariants monitoring, we are investigating sampling schemes that can reduce the monitoring cost while minimizing information loss. Most of them are based on expert knowledge about the target system, consequently the tool-set will help users obtain it.

Second, the approach generality and power could be increased by moving from invariants consisting of boolean expressions to probabilistic expressions, and by incorporating more temporal operators, which may help to capture the uncertainty present in robotic systems. It may also be worths it to integrate static analysis to guide the engine to generate more meaningful nontrivial invariants.

Last, the type of actions we support when an invariant is violated could be enriched to support, for example, message rectification so that minimally reformulated messages can be published but still remain within the system invariants. Another potential direction is learning recovery actions from users interventions. When the system misbehaves, which could help to incorporate fault tolerance into the system design.

Bibliography

[1] Ascending technologies hummingbird. http://www.asctec.de/uav-applications/research/products/asctec-hummingbird/. 5.1

[2] Claraty robotic software. https://claraty.jpl.nasa.gov. 1.1

[3] The daikon invariant detector. http://groups.csail.mit.edu/pag/daikon/. 3.1, 3.4

[4] irobot create programmable robot. http://www.irobot.com/us/learn/Educators/Create.asp 5.1

[5] Lightweight communications and marshalling. https://code.google.com/p/lcm/. 1.1

[6] Microsoft robotics. http://msdn.microsoft.com/en-us/robotics/. 1.1

[7] Nimbus lab. http://nimbus.unl.edu/projects/.

[8] Open robot control software. http://www.orocos.org/.

[9] Opencv. http://opencv.willowgarage.com/.

[10] Openrave. http://openrave.org/.

[11] Pr2 robot. http://www.ros.org/wiki/Robots/PR2.

[12] Ros. http://www.ros.org. 1, 1.1, 2.1

[13] vicon motion capture system. http://www.vicon.com. 5.1

[14] Toby H. J. Collett and Bruce A. Macdonald. Player 2.0: Toward a practical robot programming framework. In *in Proc. of the Australasian Conference on Robotics and Automation (ACRA*, 2005.

[15] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008. 2, 2.2

[16] Mauro Pezz Davide Lorenzoli, Leonardo Mariani. Automatic generation of software behavioral models. In *ICSE '08 Proceedings of the 30th international conference on Software engineering*, pages 501–510, 2008. 2, 2.2

[17] William F. Eddy. A new convex hull algorithm for planar sets. In *ACM Transactions on Mathematical Software (TOMS)*, pages 398–403, 1977. 3.4.2

[18] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, pages 213–224, 1999. 1, 1.1, 2, 2.2

[19] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, pages 35–45, 2006. 1.1, 2, 2.2

[20] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *FSE*, pages 339–349, 2008. 1.1, 2, 2.2

[21] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–254, New York, NY, USA, 2010. ACM. 2.2

[22] Franz Wotawa Gerald Steinbauer, Martin Morth. Real-time diagnosis and repair of faults of robot control software. In *RoboCup*, pages 13–23, 2005. 2, 2.3

[23] Jeremy H. Gillula and Claire J. Tomlin. Guaranteed safe online learning via reachability: tracking a ground target using a quadrotor. In *Robotics and Automation (ICRA), 2012 IEEE International Conference*, 2012. 2, 2.3

[24] Jeremy H. Gillula and Claire J. Tomlin. Reducing conservativeness in safety guarantees by learning disturbances online: Iterated guaranteed safe online learning. In *RSS*, 2012. 2, 2.3

[25] Raphael Golombek, S. Wrede, M. Hanheide, and Martin Heckmann. Learning a probabilistic self-awareness model for robotic systems. In *IROS*, pages 2745 – 2750, 2010. 2, 2.3

[26] Raphael Golombek, S. Wrede, M. Hanheide, and Martin Heckmann. Online data-driven fault detection for robotic systems. In *IROS*, pages 3011–3016, 2011. 2, 2.3

[27] UK TEX User Group. TEX Frequently Asked Questions. Available at: http://www.tex.ac.uk/cgi-bin/texfaq2html.

[28] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, 2002. 1.1, 2, 2.2, 3.4

[29] Stephen G. Hartke. A survey of free math fonts for TEX and LATEX. *The PracTEX Journal*, 1, 2006. Available at: http://www.tug.org/pracjourn/2006-1/hartke/.

[30] A. Burgin B. Zhao John-Paul. Ore, S. Elbaum and C. Detweiler. Autonomous aerial water sampling. In *The 9th Intl. Conf. on Field and Service Robotics (FSR)*, 2013. 5.2

[31] Janet L. Wiener Patrick Reynolds Athicha Muthitacharoen Marcos K. Aguilera, Jeffrey C. Mogul. Performance debugging for distributed systems of black boxes. In *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 74–89, 2003. 2.4

[32] Emre Kcman Jim Lloyd Dave Patterson Armando Fox Eric Brewer Mike Y. Chen, Anthony Accardi. Path-based failure and evolution management. In *Proceeding NSDI'04 Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 309–322, 2004. 2.4

[33] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, Chris Rowley, Christine Detig, and Joachim Schrod. *The LATEX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Reading, MA, USA, second edition, 2004.

[34] Sumant Kowshik Parth Sagdeo, Viraj Athavale and Shobha Vasudevan. Precis: Inferring invariants using program path guided clustering. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference*, ASE '11, pages 532–535, 2011. 2, 2.2

[35] Janet L. Wiener Jeffrey C. Mogul Mehul A. Shah Patrick Reynolds, Charles Killian and Amin Vahdat. Pip: Detecting the unexpected in distributed systems.

In *NSDI'06 Proceedings of the 3rd conference on Networked Systems Design and Implementation*, pages 115–128, 2006. 2, 2.4

[36] Rebecca Isaacs Paul Barham, Austin Donnelly and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04 Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, pages 259–272, 2004. 2.4

[37] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *In Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering*, pages 23–32, 2004. 5.1.1

[38] Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53:73–88, 2005. 2.3

[39] The American Mathematical Society. *The amsfonts package*. Available at: http://tug.ctan.org/cgi-bin/ctanPackageInformation.py?id=amsfonts.

[40] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, pages 282–291, 2006. 2.2

Appendix A

Grammar of Configuration File

The configuration file in XML format, where the root element is the tag ⟨*imROS*⟩. There are 21 tags at all, which are *imROS*, *scope*, *publish*, *topic*, *call*, *service*, *param arch*, *temporal*, *pattern*, *condition*, *value*, *detect*, *bag*, *check*, *success*, *fail*, *monitor*, *block*, *action*, and *violation*. Table A.1 shows the details of the grammar.

Note that some tags have the *ID* attributes, which are used to refer them in the other tags. To differentiate them, except for the *condition* tag, all *ID*s are lower case alphabet characters, and the *ID* for *condition* is a numeric character. So the attribute *events* is a string with the lowercase characters separated by commas, and the attribute *conds* is a string with the numeric characters separated by commas. The regular expression in the *pattern* tag are composed with lowercase characters referring specific events or capital characters representing any events declared in the *events* attribute.

Tag	Attributes	Elements	Comments
imROS	project	scope detect check monitor	Root element. Only one exists.
scope	-	publish call param arch temporal condition	It defines the scope of the invariant inference.
publish	id relative conds	topic param -	It defines a publishing event to be monitored. The first topic element is the monitored topic, and other topics are relative topics user declared explicitly.
topic	-	-	Its content is the global name of the topic.
call	id conds	service param -	Its elements are one service to be monitored.
service	-	-	Its content is the global name of the service.
param	-	-	Its content is the global name of the parameter.
arch	id conds	-	Its content is the architecture variables to be monitored.
temporal	id events conds	pattern	
pattern	-	-	Its content is a regular expression.
condition	id	topic value	It defines a condition in terms of the message value.
value	-	-	Its content is an expression on the field of a message or the name of another topic.
detect	inv	bag	It defines the input bag files and the output invariant file.
bag	-	-	Its content is the directory the bag file(s).
check	inv	success fail	It defines the successful runs and the failed runs used for the invariant classification.
success	-	-	Its content is the directory the bag file(s) of the successful cases.
fail	-	-	Its content is the directory the bag file(s) of the failed cases.
monitor	launch inv	block action violation	It defines how to generate the monitor.
block	-	-	Its content is the topics that the monitor will block the publishing messages if they violate any invariant.
action	id	topic service value	It defines an action in terms of publishing a message or call a service.
violation	-	-	Its content defines which action(s) will be taken at the violation(s).

Table A.1: Grammar of Configration File