Auton Robot (2017) 41:1027–1046
DOI 10.1007/s10514-016-9576-y

Inferring and monitoring invariants in robotic systems

Hengle Jiang[1] · **Sebastian Elbaum**[1] · **Carrick Detweiler**[1]

Received: 9 June 2015 / Accepted: 13 May 2016 / Published online: 30 May 2016
© Springer Science+Business Media New York 2016

Abstract System monitoring can help to detect anomalies, but crafting monitors for robot systems is difficult due to the inherent complexity, changing, and uncertain operating environment. We address this challenge by automatically inferring system invariants and synthesizing those invariants into monitors to detect faults with an approach inspired by state of the art software engineering methods. Our approach is novel in that: (1) It automatically derives invariants from messages; (2) The invariants types are tailored to match the spatial, temporal, and architectural attributes of robotic systems; and (3) It automatically classifies and synthesizes invariants into an online invariants monitor node. We have assessed the approach in the context of two unmanned aerial vehicle systems running robot operating system. We found that monitoring the inferred invariants can reduce system failure rates when facing unexpected contexts from 76 to 11 %, and can detect differences between the lab environment and the field deployments.

Keywords Invariant · Monitor synthesis · Robot Operating System (ROS) · Unmanned Aerial Vehicle (UAV)

1 Introduction

Robot systems that operate in unstructured and uncertain environments are difficult and often infeasible to test under the full range of conditions they will encounter. As such, it is relatively common to develop monitors that can detect conditions that may lead to failures and to attempt to take corrective actions. Such monitors are commonly crafted by engineers with the domain knowledge to understand what could constitute abnormal behavior. For robot systems operating in unstructured and varied environments, developing monitors becomes increasingly challenging as the system and its operating environment grow in complexity.

Consider, for example, the scenario illustrated in Fig. 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. An implementation in Robot Operating System (ROS)[1] consists of several distributed processes that communicate through dozens of message channels. An engineer developing a monitor to detect anomalies for such a UAV landing system is likely to focus on a small subset of variables and relationships between variables. For example, a monitor would likely check whether the 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 worth monitoring that are more subtle and may not be considered by the engineer given the number of variables and relationships involved and the wide range of operating conditions. For the UAV landing system, for example, it would be useful to have monitors to check whether 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, or that changes in the UAV internal Inertial Measurement Unit (IMU) are followed by changes in the external localization system. The goal of this work is to support the automatic generation of such invariants and the associated monitors.

✉ Carrick Detweiler
carrick@cse.unl.edu
http://nimbus.unl.edu

[1] Nebraska Intelligent MoBile Unmanned Systems (NIMBUS) Lab, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 220 Schorr Center, Lincoln, NE 68588, USA

[1] Robot Operating System (ROS), http://www.ros.org.

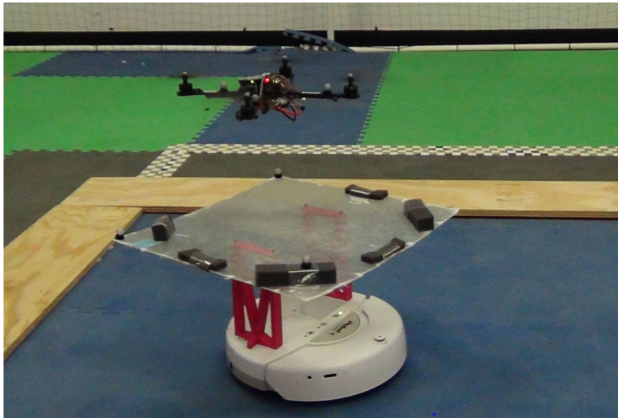


Fig. 1 UAV attempting to land on a moving platform

1.1 Approach overview

In this article, we propose an approach to automate the synthesis of invariant monitors for robotic systems through the analysis of their traces. Our approach is inspired by existing software engineering approaches for automated invariant inference (covered in Sect. 2.2). 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 invariants that are violated by other trace values.

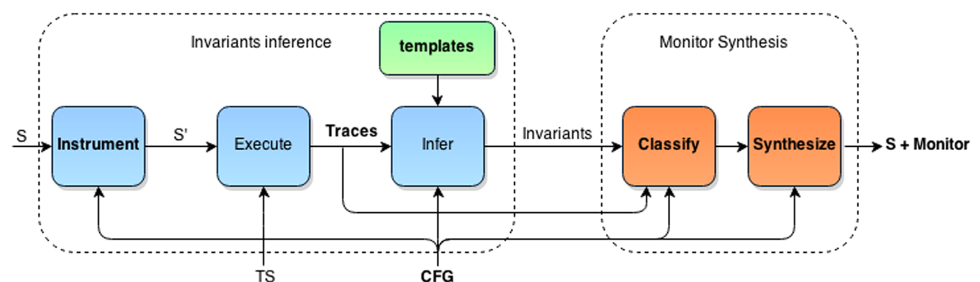
Existing techniques to automatically infer invariants have been shown useful for generating invariants that serve as pre and post conditions for functions. The application of these techniques to large distributed robotic systems, however, has been limited. We conjecture that this is due to the focus on the generation of low level invariants, which is impractical for these large systems. In addition, these systems do not take into account the physical nature of robot systems and therefore lack domain-specific invariants that capture the temporal and spatial aspects of robotic systems, Finally, these tools cannot be easily integrated into distributed robot systems, such as ROS, which is used in a large number of robot systems. In this work we set out to tackle these challenges.

At a conceptual level, the proposed approach first infers likely invariants from successful runs collected in traces. Then, it automatically synthesizes a monitor that is inserted into the robotic system as a ROS node to detect invariant violations in future runs. When there is an invariant violation, the monitor will log it or take a user-specified recovery action ranging from blocking messages to executing a new control action. These recovery actions must be carefully designed to not cause additional failures, and their associated tradeoffs analyzed. This can be challenging in the general case, but reasonable for specific contexts. For instance, in the UAV landing scenario, aborting a landing is a typically safe action that is preferable to a failed landing. The recovery actions can also be tied to the software state so that the UAV, for instance, does not try to abort a landing when it is searching for the landing platform.

Figure 2 shows the workflow of the approach, which differs from existing dynamic invariant inference frameworks (e.g., Ernst et al. 1999, 2006; Gabel and Su 2008; Hangal and Lam 2002) in the areas with bold labels. The system only requires three inputs: the target system S, a configuration file CFG, and a training set of activities TS. In the context of ROS, the system S consists of the nodes and launch files that make up the robot system. The configuration file, CFG, is a custom XML-file that defines the ROS-topics to examine and the safe actions to take in case of an invariant violation. Finally, the training set of activities, TS, are the activities to be executed by the robot (e.g., landing a UAV on a moving platform) that serve as a model of correct behavior.

In the invariants inference block of Fig. 2, the target system S is automatically instrumented to capture the messages passed between the ROS nodes in the system. We call the instrumented system S'. When S' is executed on the different training activities, TS, a set of $Traces$ is generated, where each trace will contain a sequence of variable-value pairs found in the system communication messages. The approach then attempts to instantiate the invariant templates tailored for robotic systems or to refine already generated invariants based on the information found in $Traces$. The invariant templates used and the topics to be examined in the traces are defined by CFG. The monitor inference block starts by classifying the generated invariants to prune the less useful ones. The final output is the system

Fig. 2 Approach workflow



with modifications that include an extended ROS launch file that remaps the system messages to pass the needed data through an automatically synthesized monitor node in order to detect invariant violations and launch countermeasures.

While we have implemented our approach for ROS, we note that it could also be implemented on many other robotic systems made of distributed nodes that sense, actuate, and communicate through some form of message passing scheme (e.g., LCM,[2] Microsoft Robotics Developer Studio[3], CLARAty[4]). Doing so would require tailoring certain lower-level aspects associated with instrumentation, trace generation, and monitor generation, that are often coupled with each middleware. In addition, the approach may not be applicable to components of robot systems that have hard real-time constraints where adding delays due to the monitor or corrective actions could violate real-time constraints. Systems such as ROS, however, operate at a higher level and do not make any real-time guarantees so the approach is appropriate for many ROS systems.

In this work we test our approach on two case studies. The first corresponds to the system shown in Fig. 1, the UAV landing system. We found that with the synthesized invariants monitor the UAV landing system was able to handle situations that it was not designed to handle, such as when the landing platform was already occupied, when there were additional fake landing platforms, when there was a strong wind, and when the platform was broken. The second is a UAV-based water sampling system from an unrelated project that has conducted hundreds of missions in the field (Ore et al. 2015). In this study, we found that a monitor encoding invariants generated with traces collected on indoor trials showed a number of violations when tested outdoors due to differences in sensor rates and the outdoor environment. This led to a refinement of the indoor test environment to better match the outdoor environment. In addition, it led to the discovery of a subtle bug (under particular conditions, the pump was turned on without water) that was not discovered in hundreds of field tests, but probably contributed to the early breaking of several pumps. These case studies illustrate the power of automatically generated monitors for robots.

1.2 Contributions

This article is a significant extension and revision of our prior conference article on inferred invariants (Jiang et al. 2013). Specifically, we provide additional detail on the invariant generation and new invariant templates in Sect. 4. We have also extended our monitor generation approach to statistically prune the set of invariants based on positive and negative tests (Sect. 5.1), provide details on our monitor synthesis (Sect. 5.2) and configuration methods (Sect. 5.3). Finally, in Sect. 6, we evaluate the performance of the approach in more detail on the UAV landing scenario and on a new UAV water sampling case study.

Our main contributions are the:

- Development of novel invariant templates that account for critical properties in 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 (e.g., publish ordering of different message types).
- Capture and processing of traces through high-level and lightweight system instrumentation (just the message passing channels). Operating at this granularity enables the system characterization of the behavior of a system's nodes or families of nodes, with the corresponding generation of higher-level invariants.
- Implementation of a version of the approach that automatically classifies the quality of the inferred invariants based on their failure prediction power and synthesizing those invariants into a monitor node that can be seamlessly integrated into existing ROS-based robotic systems. The monitor can be tailored to trigger actions when an invariant is violated.
- Assessment of the approach in the context of two UAV systems to better understand its potential. Our experiments revealed that the monitor can reduce system failure rates when facing unexpected scenarios from 76 to 11 %. In addition, the inferred invariants can be used to check changes in deployment conditions and developer's expectations.

2 Background

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, we first introduce and discuss details of this robotic system and use it to convey the elements of similar architectures. We then give background on Daikon, the invariant inference engine on which we build.

[2] Lightweight Communications and Marshalling (LCM), https://code.google.com/p/lcm/.

[3] Microsoft Robotics Developer Studio http://msdn.microsoft.com/en-us/robotics/.

[4] CLARAty Robotic Software, https://claraty.jpl.nasa.gov.

2.1 ROS

ROS is a software framework for robot software development, which provides operating system-like functionality on a heterogeneous computer cluster. Nodes are the basic elements, each being a process launched under ROS, and communicating with each other through topic messages and services. A topic works like a message bus, where nodes can publish to and subscribe from; the messages published on topics are user defined data structures. Multiple nodes can publish or subscribe to one topic. In contrast, a service can only be provided by one server, and the client nodes can communicate with the server node through a call. All these communications are registered and directed by a master server node.

ROS also provides several ways to configure the system at launch time through an XML-format launch file. This adds flexibility by enabling the restructuring of the publisher-subscriber and service architecture and the redefinition of parameters without changing the source code. Similarly, at runtime, the master node contains the ROS parameter server which provides a dictionary-based representation to access and change parameters that are used by the nodes at runtime.

Our approach targets all these ROS elements, analyzing and monitoring the data in messages, services, and parameters, and manipulating launch files to transparently implement the monitoring process.

2.2 Invariant inference in daikon

Our work was inspired in part by the evolution and maturity gained by techniques and tools available to infer likely program invariants. Our toolset builds specifically on daikon (Ernst et al. 1999, 2006), one of the pioneer approaches with probably the most sophisticated toolset openly available.[5] The invariants produced by daikon are an analogue of a low level program specification in the form of a method pre- or post-conditions over the method parameters, such as $x! = null, y > 0, x \quad mod \quad y = 0$, and $array[] = \{1, 2, 3\}$. Since these invariants are generated based on the data available in program traces, they can only characterize the behavior present in those traces. This means that the inferred invariants may not always hold as they can under-approximate the program behavior (when certain behavior is not exposed by trace) or over-approximate it (when the trace lacks the behavior that would invalidate an inferred invariant). In spite of these limitations, such invariants have been used extensively as part of many software testing, debugging and verification techniques.

Daikon provides several language-specific front-end tools for program instrumentation and an extensible invariant template set. Front-end tools insert probes into the target program (more specifically at methods' entries and exits) to enable the capture of variable-value pairs at those particular execution points. They also generate a *.decl* file containing the variables captured at each program point. During execution, these probes output variables' values to the data traces in daikon's format ($.dtrace$ files). Next, at inference time, the invariant templates are used to initialize and check for invariants on the data traces, and developers can extend this template set. The inference engine follows these steps: (1) Given variables declared in *.decl* and daikon's configuration parameters, it initializes all possible invariants on the variables based on the invariant templates. (2) It reads the data from the $.dtrace$, checks all the invariants initialized in the first step, and dismisses or refines them if the data in the trace violates them; This step is repeated until the end of the trace. (3) After finishing reading the trace data, it filters out unjustified or redundant invariants based on the settings, and finally outputs the invariants remaining. For example, given a template invariant $varX > constant$ and a trace of six variable-value pairs collected from time $t1$ to $t6$, $tr = [t1 : a = 1; t2 : b = 3; t3 : a = 1; t4 : a = 2; t5 : a = 1; t6 : a = -1]$, daikon would instantiate the invariant template as $a > 0$ after reading that the value of a at $t1$ and then at $t3$ are greater than zero, but then refine this invariant to $a > -2$ at $t6$ when value $a = -1$ is observed; for variable b an invariant may be inferred but not reported as there may not be enough values to support that instantiation. Our approach extends the front-end, the set of invariant templates, and it incorporates a new component to perform further invariant refinement and monitor synthesis.

3 Related work

In this section we explore the related work in two contexts: invariant detection and monitoring, and robot execution monitoring and distributed debugging.

3.1 Invariant inference and monitoring

In the context of daikon, our contribution provides a new front-end that operates at conceptually different program points that are associated with the message publishing or service calls, new invariant templates specially evolved for robotic systems, and the incorporation of those invariants into a monitoring node. Before we delve further into our approach, we further discuss other related approaches to invariant inference.

DIDUCE (Hangal and Lam 2002) was among the first to use invariants for runtime monitoring and diagnosis, which

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

is something we are doing as well with the invariants we infer. DIDUCE instruments java bytecode, focuses on program states at particular program points (procedure calls and heap accesses), and relaxes invariants or reports them to users when detecting anomalies. Instead, our approach operates in the context of message-passing and service-oriented architectures supporting distributed robotic systems, does not instrument the program source code, incorporates domain specific invariants, and can take corrective measures (like interrupting certain messages) to prevent system crashes.

Several other complementary efforts have emerged in the last few years, ranging from those attempting to refine state invariants to those attempting to infer richer invariants (Gabel and Su 2008; Yang et al. 2006; Csallner et al. 2008; Sagdeo et al. 2011; Beschastnikh et al. 2011). To improve invariant generation, researchers have taken advantage of static analysis to guide dynamic invariant inference. For example, PRECIS proposed generating invariants through program path guided clustering (Sagdeo et al. 2011). 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 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 (Csallner et al. 2008). In our work, we did not apply static or symbolic analysis because of the scale as we are facing large scale distributed robotic systems instead of a class or a function. Consequently, our approach is based on the analysis of traces without any dependence on source code.

The body of work on specification patterns (e.g., Autili et al. 2015; Dwyer et al. 1999; Ghezzi and Kemmerer 1991; Grunske 2008; Konrad and Cheng 2005) that define a rich set of operators to characterize a broad set of temporal, real-time, and probabilistic behaviors is extensive. The work associated with automatic inference of such patterns, however, has been much more limited and mostly limited to temporal invariants that encode simple rules on events' order. Javert 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 (Gabel and Su 2008, 2010). We have implemented a subset of such temporal invariant inference on published messages and called services. We specifically infer the ordered-pair interval invariant, which states that an event always happens after another event within certain time and events in between. Some follow up efforts on invariant detection also aimed to characterize higher level of abstractions. Henkel and Diwan (2003), for example, derived algebraic class specifications, while Beschastnikh et al. (2011) derived models such as call graphs. Most of these efforts focused on interactions between components, and the results are often in the form of finite state machines (FSM) or call structures. Lorenzoli et al. (2008), for example, developed a dynamic analysis algorithm called GK-tail combining the ideas of invariant detection and temporal property mining to produce an even richer extended finite state machines (EFSMs). We leave the incorporation of these extensions to future work.

3.2 Monitoring and debugging robotic systems

In the context of robotic systems, monitoring for error detection is a well known area (Pettersson 2005). 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 broadly categorized into model-based or data-driven, based on how they build the system model (invariants in our approach) to detect anomalies.

With model-based approaches engineers model each state and use this model to estimate the current system state (i.e., normal or faulty). They are commonly used, for example, when designing control systems, where precise models target problems fairly close to the hardware as well as the raw sensor data. In the context of quad-rotor UAVs similar to the ones we used in our study, there have been several recent efforts that attempt to detect specific known anomalies by model-based approaches. For example, Gillula and Tomlin (2012) proposed a framework using reachability analysis in a way that prevents the control system from taking an unsafe action. Muller and Sukhatme (2014) generate risk-aware trajectories for landing a quad-rotor in the presence of obstacles. Sattar and Dudek (2014) develop a model of human-robot interaction to minimize risk while taking into account task specifications, communication, execution costs, and other factors with an underwater robot.

The data-driven approach does not need an a priori model; instead it tries to infer an abstract model (usually through a statistical analysis) of the original system from the data, and it uses the inferred model to detect faults. For example, Golombek et al. (2010, 2011) presented one example of such a model that works on message-passing robotic systems, mapping each system's internal data exchange between component to an event. Then, it infers a probabilistic model on the event sequences, which is a histogram of the interval time between any two events, by which it could detect component failures, resource starvation, and asynchronous communication errors. Mendoza et al. (2012) build a Hidden Markov Model by observing the ground robot over time and then uses this data to develop a monitor to detect collisions and other faults. Our approach also infers one of such temporal invariants as the ordered-paired interval invariant, which

captures not only the time but also the events happening in the interval.

There exist efforts aimed at diagnosing and remediating anomalous behavior based on models defined by domain experts. Steinbauer et al. (2005), for example, presented a solution for real-time fault detection and repair of control software of autonomous robots. Their diagnosis system uses model-based diagnosis for fault detection and localization, and a repair module that executes predefined actions to recover the system from the fault.

Our approach is complementary to these approaches and was first presented in (Jiang et al. 2013). In our earlier work, we initially presented the concept of inferring and monitoring invariants for robot systems and evaluated it on a UAV landing system, however this early work had a more limited set of invariants and a synthesizer that included manual steps. In this work, we extend the invariant templates and generation technique to more general invariants that were not considered by domain experts, automatically prune the set of invariants, provide additional details on the automated configuration and monitor synthesis, and evaluate the approach on two case studies. The approach is implemented within ROS, which makes it directly applicable to a large set of existing robotic systems.

Although with the broader goal of facilitating robot programming, the graphical state-space programming interface (Sattar et al. 2010; Li et al. 2011) also uses invariants. Contrary to our approach, invariants are not inferred, but rather used to determine the actions available to a user at a particular program state defined by satisfied invariants. Such approach could be used to enrich the monitors synthesized by our approach if a user is meant to be a part of that loop.

In the context of distributed system debugging, Reynolds et al. (2006) state that developers usually focus on two kinds of bugs: structure and performance. A structural bug results in processing or communication happening at the wrong place or in the wrong order. Most debugging approaches for networked distributed systems (e.g., Reynolds et al. 2006; Barham et al. 2004; Chen et al. 2004) tend to collect event sequences as causal paths, and check those sequences against expected properties. Along similar lines, our approach captures temporal and architecture invariants (see Sect. 4.2.3). A performance bug, for example computational time or communication bandwidth, results in processing that consumes too much or not enough of some important resources. Our approach also infers some invariants of system performance such as message frequency and delays.

4 Invariant inference

In this work we aim to capture invariants that are more specific to robotic systems. We do this through the development of an invariant inference and a monitor synthesis components that enable the generation and checking of new invariant types (illustrated in Table 1) that reflect the spatial, time, architectural, and temporal nature of robotics systems. We now proceed to describe each component in more detail.

4.1 Trace generation and translation

The goal of trace generation is to collect enough data into traces to infer the target invariant types. Because of the invariants we aim to generate, we want each record in the traces to include the time stamp, the message type, the message values, and the message topic. The existing ROS's rosbag tool[6] is able to meet these requirements for topics but it lacks information about services, ROS parameters, and ROS architecture, so we need to enrich the default ROS traces with the required information. The basic mechanisms we use to enrich traces consist of remapping the topics and services using the standard ROS remap command and adding a ROS node that we call the "recording node," which publishes the missing information.

To record service usages, our approach first queries the ROS master node at the beginning of execution to get the list of all the available registered services. Next, it remaps each original service to the recording node, which publishes each transaction on a regular ROS topic that can be logged with rosbag. In addition, the recording node creates a new service that relays the request and response messages between the client and the original server. For illustration, consider the system at the top of Fig. 3. In the original system, the client node calls a service named $/s1$. As illustrated at the bottom of Fig. 3, our approach remaps the original service to $/rec/s1$, and makes the recording node relay the service. Then, 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 it all into the trace. As we shall see in Sect. 6, service relaying does introduce a small delay, which is on the order of Ethernet network latency.

Figure 4 illustrates how our approach captures parameters and architectural information. To capture global system parameters, the recording node queries the ROS master to get the parameters set on the server, and then publishes them to the newly incorporated "parameter" topic. The query occurs at configurable time intervals (typically on the order of every second), only publishing changed parameters to minimize redundant messages. Similarly, the architectural information is collected by the recording node through a series of queries to the ROS master to retrieve publishers, subscribers, service providers, and serviced clients. The recording node takes a snapshot of the architecture at a con-

[6] ROS Bag utility, http://wiki.ros.org/Bags.

Table 1 New invariants templates

Invariant type	Characterization	Formulation (constants abbreviated by cnst)	Examples (from the case studies)
Time-related	Topic and message's content frequency, variance, and change rate	Given $topic$ with messages $\overline{m}$ containing variable v, and a window size w, compute: $topic_frequency$, $m.numVar_variance$, and $m.numVar_changeRate$, to derive:	On the frequency of the iRobot topic:
		$cnst.lwr \leq topic_frequency \leq cnst.upper$	$iRobot_frequency > 2.05Hz$
		$cnst.lwr \leq m.v_variance \leq cnst.upper$	On the rate of change of the iRobot location on the x-axis:
		$cnst.lwr \leq m.v_rate \leq cnst.upper$	$-0.45 <= iRobot.x_rate <= 1.01$
Polygon	Relationships between two variables	Given variables x and y: $\cap_i^n(cnst.a_i x + cnst.b_i y + cnst.c[>= \mid <=]0)$ where $cnst.a$, $cnst.b$, $cnst.c$ are constants	On the area defined by the UAV.location with respect to iRobot.location: $UAV.y + 0.0554 * iRobot.y > 1.89 \cap$ $UAV.y - 0.990 * iRobot.y < 0.151 \cap \ldots$
Architecture	Inter-node communication graph capturing messages and services	Given resource R (topic or service):	On the nodes subscribed to UAV.pose (a single node in this case):
		$R : (max_node_set, min_node_set)$	UAV.pose = ({vicon},{vicon})
Temporal	Temporal properties associated with pairwise events' sequences	Given events A and B:	On the temporal dependency between IMU and POSE messages:
		A is followed by B or $A \rightarrow B$ $((\bar{A}^* A \bar{B}^* B)^+)$:	$IMU \rightarrow POSE$: {0.00005, 0.0621, 2, 2}
		$\{minTime, maxTime, minEvents, maxEvents\}$	

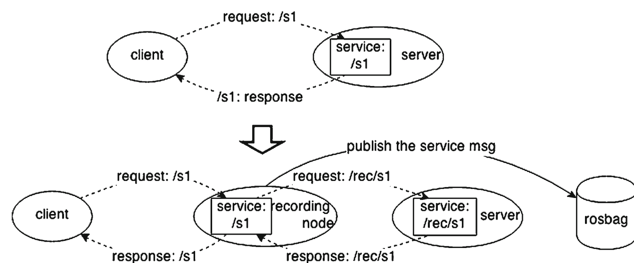


Fig. 3 Example of remapping service

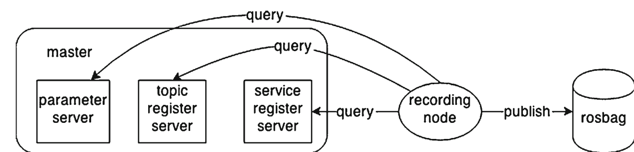


Fig. 4 Architecture and parameter recording

figurable time interval, and it reports revised architectures. 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: $\{service1 : server1, service2 : server2, \ldots\}$.

Once traces are collected, the approach performs a translation into daikon's format to later enable the generation of the target 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 the variables available at the same program point are analyzed together to compute invariants. For example, daikon would attempt to infer the relationship between variables $x1$ and $x2$ at the entry point of method A, but it would not attempt to make inferences between $x1$ or $x2$ at the entry point of method A and the variable y at entry point of method B.

For the systems we are analyzing, capturing traces at the method or even at the class level would generate too much and often not very meaningful data. So instead, our approach clusters topic messages consumed and published at the node level to better capture the behavior of the overall system. The key insight is that the entry values in the messages consumed by a node are likely to define its behavior, affect its outputs, and become evident in the published messages. To leverage that insight, for a target node, we pair each published message with the messages in each subscribed topic.

For example, in the ROS system shown in Fig. 5, each ellipse represents a node, rectangles show topics, and the

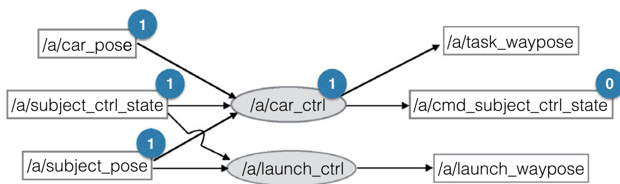


Fig. 5 Program points in a ROS system

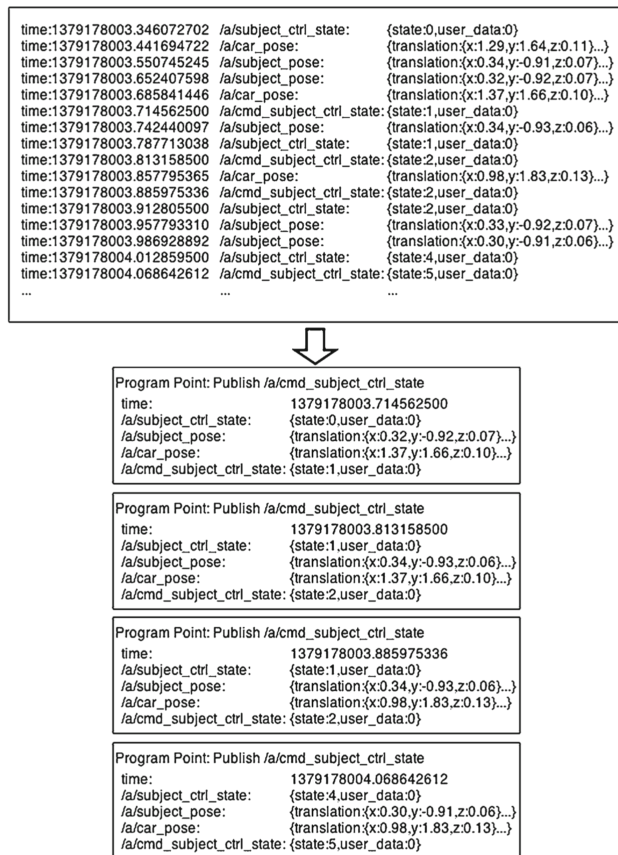


Fig. 6 Sample trace (*top*) and message pairings (*bottom*)

directed edges tell the publish and subscribe relations. At the center of the graph is the node $/a/car_ctrl$①, which publishes to the topic $/a/cmd_subject_ctrl_state$⓪ and subscribed to $/a/car_pose$①, $/a/subject_ctrl_state$① and $/a/subject_pose$①. The top of Fig. 6 contains a sample trace that includes a time stamp, the topic, and the message. The bottom of the figure shows the result of translating that trace around the program point $/a/car_ctrl$. The clustering of messages around the program point is performed by pairing each published message on $/a/cmd_subject_ctrl_state$, with the latest values of messages published in the subscribed topics $/a/subject_ctrl_state$, $/a/car_pose$ and $/a/subject_pose$. The translation of this trace results in four pairings for the chosen program point.

For services, the translator retrieves the messages (request and response) on the service calls from the service-recorded topic, and groups these messages into corresponding service program points. For architectural and parameter messages, our approach reads and converts them into the data trace format and represents them as string variables. The trace generator and translator's operating options can be set through the configuration file, including what messages to monitor, what type of data to collect, and the length of the program point message chains.

There are a number of additional considerations in this process. First is the number of messages considered in the pairing. In our implementation we only considered the last messages on each subscribed topic before publishing a message because we assume that the newest information is the most valuable. Considering multiple messages may be useful to better characterize nodes significantly affected by intermediate messages. That would require, however, more complex and expensive invariants to take advantage of that information. Second, our analysis as described only targets one node. But targeting multiple nodes by clustering may be of interest to form a richer chain of potential influence. Increasing the chain of potentially influential publishers may capture new interesting invariants, but it also increases the size of the program point making invariant generation and monitoring more complex and expensive, and it is also more likely to report accidental relationships.

4.2 Invariants types

The goal of the invariant inference is to generate invariants that hold for all trace data. The process is depicted in the pseudocode in Algorithm 1. It consumes the $msgsPairings$ produced by the trace generation and translation process described in Sect. 4.1, and the set of $templateInvariants$ that we have illustrated before and will describe in more detail next. The algorithm iterates over each message pairing $mPair$ (line 3), and each invariant template (line 4), analyzing whether it is part of an existing invariant or if it has the potential to create a new invariant (e.g., message on a new topic or on an unseen pair of topics, message with new event). If it is part of an existing invariant, then the $mPair$ data is used to revise the invariant (line 6) either by refining it (e.g., by relaxing the bounds) or by dropping it altogether (e.g., if the temporal relationship between two events no longer holds). If it is not, then a new invariant (line 8) is instantiated using the data in $mPair$ (and in previous $mPair$s as needed depending on $tInv$).

The process we used to derive these particular invariant templates was incremental, starting with the ones provided with daikon, and evolving them to capture properties that are

often used in the specifications of robotic systems. This evolution took advantage of a multi-year collaboration and the mixed expertise of the co-authors in robotics (to suggest properties that may be generally useful) and program analysis (to assess their feasibility in terms of cost and precision). In the end, the invariant types reflect three distinguishing attributes of robotic systems: they operate under time constraints, they operate under physical space constraints, and they react to the user input and the sensed environment. In addition, when implemented on a dynamic publish-subscriber middleware, particular architectural dependencies emerge. Our invariants types reflect each one of those attributes. We discuss each type in detail next.

Algorithm 1 Invariant Inference

```
1: function INFER(msgsPairings, templateInvariants)
2:    inferredInv ← ∅
3:    for each mPair in msgsPairings do
4:       for each tInv in templateInvariants do
5:          if inferredInv.EXIST(tInv, mPair) then
6:             inferredInv.REVISE(tInv, mPair)
7:          else
8:             inferredInv.INSTANTIATE(tInv, mPair)
9:          end if
10:      end for
11:   end for
12:   return inferredInv
13: end function
```

4.2.1 Time-related invariants

In robotic systems, the program state is intrinsically defined not just by the variables' values, but also by how those values change over time. This new type of invariant template leverages the program points' time-stamps to characterize messages and their rate of change.

The simplest of the new templates serves to characterize messages' frequency and variance. For frequency, this takes the form of $cnst.lwr \leq topic_frequency \leq cnst.upper$. For the UAV landing system detailed in Sect. 6.1 and Fig. 1, this type of invariant is useful to detect, for example, stale location data that may direct the UAV through an erroneous navigation pattern when the communication is broken. The variance invariant takes a similar form as $cnst.lwr \leq m.v_variance \leq cnst.upper$, and can help detect data that does not fit a given distribution.

A more complex time-related 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 $cnst.lwr \leq m.v_rate \leq cnst.upper$. In our scenario, a common instance of such an invariant is $minVelocityUAV \leq UAV.distance_rate \leq maxVelocityUAV$, which can detect wrong localization data as it will be shown in our case study.

Computation: To infer these invariants, we take advantage of the time stamps attached with the observation at the program points. We added these three new invariant templates in daikon to associate the time component with variables at a program point. The computing procedure uses a moving window (its size parameterizable) on a trace file and the associated time-stamps to compute frequency, variance, and change rate of all variables present at a program point. Then, the minimum and maximum values for frequency, variance, and change rate observed across the whole trace are instantiated as invariants. The procedure complexity is a function of the trace and window sizes. In practice using window sizes of 5–10 elements was sufficient for messages published at 10s of Hz, but these may change depending on the variability of the observed variables.

4.2.2 Polygon invariants

To reflect the notion of spatial bounds, we introduce invariant templates that define relationships between two variables[7] that can be characterized through a convex polygon. For example, if our operating scenario was bounded by the dimensions of a room, this invariant template would ideally be instantiated and refined into a polygon similar to the shape of the room. 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.

Computation: To infer this invariant, 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, else if the observation is within a specified distance from the polygon, the polygon is relaxed by computing the convex hull that includes the new observation (Eddy 1977), otherwise it is discarded. We note that this type of invariant can also characterize relationships between variables that are not spatial per se. Consider 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.

4.2.3 Architecture invariants

Distributed robotic systems have a dynamic architecture that can be tweaked for different deployment conditions. When misused, this flexibility can often lead to erroneous con-

[7] The cost of generating invariants with more than two variables was exponential and hence prohibitive unless it was focused on a small set of topics.

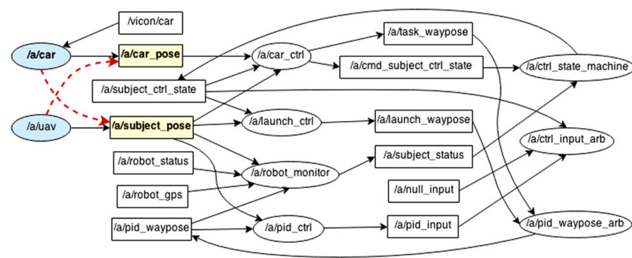


Fig. 7 A ROS system. Wrong deployment setting represented with *dashed lines*

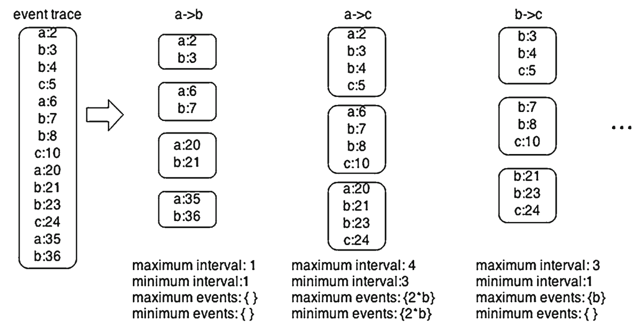


Fig. 8 Event trace and interval analysis

figurations, causing additional or missing topics or nodes. Take the ROS program shown in Fig. 7 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 spawned from the same source code but with different remapped names, it is easy for users to create mappings that cause incorrect connections as shown with the two dashed lines in Fig. 7. Although the messages' values may look correct, the system could be operating on the wrong messages, and be generating the wrong control inputs for the UAV and the car. The architecture invariants could help detect such problems. They take the form of the maximum and minimum nodes set using some particular communication resources as $\{resource : (max_node_set, min_node_set), ...\}$. For Fig. 7, for the topic $/a/subject_pose$, the invariant of publishers is $/a/subject_pose : (\{/a/uav\}, \{/a/uav\})$, which means that there must be only one node named $/a/uav$ publishing to this topic. If the system is launched with incorrect mappings as shown by the dashed lines, this architecture invariant will be violated, because $/a/uav$ does not publish to topic $/a/subject_pose$ and $/a/car$ publishes to this topic.

Computation: To infer these invariants, we take advantage of the richer trace we generated to include information about the system architecture. We analyze the trace records on the architecture topic, updating the minimal and maximal set of subscribers and publishers per topic, and the clients on each service. To update the maximal set, we perform a union operation between the existing set of publishers and subscribers and the ones in the incoming record. To update the minimal set we perform the intersection.

4.2.4 Temporal invariant

A temporal invariant expresses order of events' sequences. For example, in our UAV landing system, the *pid_ctrl* node should not publish control messages until it gets the iRobot and the UAV position messages, a landing event should always be followed by a decrease in thrust, and a moving forward event should always follow a pitch command. We have focused on a temporal invariant inference template to capture such sequences called ordered-pair interval. It expresses that an event A is followed by event B, with other events interspersed as long as they are not A or B. More precisely it expresses $(\bar{A}^*A\bar{B}^*B)^+$. It can also specify the interval between events in terms of time or number of events. This invariant captures the minimum and maximum time and number of events in the interval.

Consider the trace of events a, b, and c, and their times as shown in the first column of Fig. 8. Looking first at just the ordered-pair ab, the inference engine finds four instances of the pattern, presented in the second column. It then computes the interval invariants shown at the bottom of this column. The first two invariants are the maximum and minimum interval between the events, which indicate that once a happens b should happen within 1 unit of time. The maximum and minimum event sets in the interval are empty, which means b should follow a without any other events in between. So, event sequences like acb or aab would violate this invariant. From the same trace, it can infer similar invariants for other event pairs like ac (third column) or bc (fourth column).

Computation: To infer this invariant the approach initializes all potential ordered event pairs. Upon reading an event from the trace, it updates all the ordered-pair intervals. If the interval time is larger than a configurable threshold , or the size of the event set is greater than another configurable threshold, we remove this interval or set because we interpret it as a weak relationship. In practice, the interval times we use are relatively small (on the order of tenths of a second) since most events in the systems we analyzed are tightly coupled.

5 Monitor synthesis

Now that we have discussed how we analyze system traces to infer new types of invariants, 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 that will be integrated into the system to check the inferred

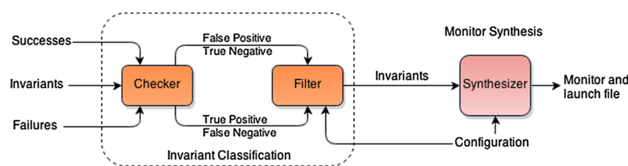


Fig. 9 Monitor synthesis work flow

invariants automatically. As shown in Fig. 9, the workflow involves two components, *InvariantClassification* and *MonitorSynthesis*, that we now describe in more detail.

5.1 Invariant classification

During the inference process, invariants that are statistically justified are outputted, while the rest are dropped. This pruning procedure brings at least two benefits. First, it reduces the monitoring overhead of checking those weaker invariants, which can be significant. For instance, in the case study in Sect. 6.1, checking 1206 polygon invariants adds an average of 1.4 ms latency. Second, because of the insufficient samples of system's behaviors exposed by the traces, some invariants tend to under-approximate the system behavior, and hence are too fragile. For example, as we shall see in Sect. 6.2, the polygon invariant inference generates about n^2 polygons for 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.

Our approach provides an extra layer of pruning the set of inferred invariants, as shown in Fig. 9, testing the invariants against passing, but also failing traces, to understand the value of the generated invariants to detect failures. In a sense, we treat each invariant as a binary classifier of system and measure the performance of these classifiers. Intuitively, if an invariant is not violated in any successful run but it is violated in all failed runs, then it is an ideal classifier with the highest performance to detect faults. On the other hand, if an invariant is violated by both the successful and failed runs, then it may be a poor classifier. To do this classification, we check each invariant to compute the true negative and false positive rates using successful traces, while we get the true positive and false negative rates from failed runs. Since it is convenient to have a single value to score each invariant, we also compute the *precision*, *recall*, and the *F-score* (Witten and Frank 2005). Last, we use these computed values to determine what invariants to filter. The implementation also allows for the *Checker* component to work independently of the filter in order to allow for comparisons between different data traces. This is convenient, for example, to compare the effects of different environments in the generated invariants. We will see the application of this in Sect. 6.2.

5.2 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 monitor node that checks whether the system violates the inferred invariants. We have three sources of data that need to be monitored: messages on topics, messages relayed by services, and parameters and architecture information that need to be queried from the ROS server.

For messages on topics, the monitor simply subscribes to the desired topics, and every time it receives a message it computes the necessary additional variables (e.g., message frequency, variable variance, change rate), checks the corresponding set of invariants, and finally reports if any of them are violated by the message. Since these invariants are boolean expressions, such checks are quite straightforward. For messages coming from a service, the monitor operates in a similar fashion except that it needs to intercept the service communication as per the ROS mechanism. For parameters, the monitor queries the ROS master every specified interval (typically on the order of every second), and does the same check as it does to other invariants. For the architecture invariants, the monitor also queries the master node every specified interval to get the architecture information, then it checks if any nodes are within the inferred sets. To check order-paired interval invariants, the monitor keeps a timed trace per event-pair. When the first event of a pair is observed, the intermediate events are recorded until the second event of the pair is observed. At that point the set of events is compared against the inferred ones to check for violations.

The monitor also encodes the actions to be taken if an invariant is violated. The recovery actions our approach currently supports are shown in Table 2, including raising a warning, blocking messages causing a violation, publishing a message, calling a service, and unregistering an unknown node. The default action is raising a warning. We leave it to the developer to determine if one of the other recovery actions is appropriate to take the system to a reachable safe state. For instance, if the code being monitored sends a message to another component of the system to start an action, then it may make sense to block this. Or if there is a control

Table 2 Supported recovery actions

Recovery action	Relevant invariants	User defined
Raise a warning	Any	No
Block bad message	Topic and service	Yes
Publish a message	Any	Yes
Call a service	Any	Yes
Unregister unkwn pub	Architecture	No

action that is safe under a wide range of settings, then that action could be initiated if there is a violation.

We acknowledge that developing cost-effective recovery actions can be challenging. Particular care must be taken to ensure that the recovery action itself does not lead to additional failures, or to more costly failures. This is especially important since there will likely be false positives where the monitor detects a violation that would not necessarily lead to a failure. As discussed, a richer training data set can help by reducing the number of false positives and also by informing the developer on the contexts under which inferred invariants are checked. But ultimately the developer must make an assessment of the likelihood of safely transitioning to a target state, and the likelihood that that target state is safe. In practice this process often entails exploring that space first at design time and then also during development. For example, during our project development we initially activate the warnings for all violations (default mode), and then incrementally identify violations or classes of violations leading to failures with enough frequency and cost that taking a corrective action becomes an enticing, cost-effective proposition. Throughout this process we also identify particular contexts under which such violations should be assigned different weights. For instance, we configured the system to only execute an abort landing command if the system is currently in the landing configuration and only produce invariant warnings during translation periods. In the end, the developer must make an assessment of whether the likely costs and benefits associated with performing a no-action is preferable to performing an action. In this work we have found that making general assessment is challenging, but there are many particular instances for which it works remarkably well.

5.3 Configuration

The whole system for inference and monitoring can be configured through a XML-format configuration file. A brief example is shown in Fig. 10. Under the tag *scope* the elements to be recorded in the trace are specified (two topics, a service, an architecture, and a temporal pattern). The location of the bag trace and the invariant generated are specified under the *detect* tag. 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. Under 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. Under the *action* tag a particular message to a topic will be published as defined by the *value* tag. The *violation* tag specifies the program points at which those actions will take place if a violation occurs.

```
−<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>
```

Fig. 10 Configuring the approach

In the *monitor* tag of the configuration file, the action is defined under the *block* tag and works for all state invariants on messages on topics and services. The user can define on which topics or services the monitor is going to drop the messages if invariants are violated. In Fig. 10, the monitor will block the "bad" messages on the topics */a/cmd_subject_ctrl_state* and */a/task_waypose*. Thus, for these two topics, the messages are not just consumed by the monitor, but also intercepted and only republished if they do not violate any invariant. The monitor can also block a service. These kinds of actions can be useful in preventing the system from getting into an abnormal state driven by the "bad" messages. The monitor can also publish a particular message on some topic or call some service with some arguments. These actions are defined in the *action* tag. These actions can be applied to any kinds of invariant violations by declaring them in the *violate* tag. For example, in Fig. 10, the action labeled with f will be taken when any invariant is broken on the program points labeled as a, c, e.

6 Case studies

In this section, we present two case studies to assess our approach and explore its potential.

In the first case study, we explore if the invariant monitor with associated actions can reduce a system's failure rate, and how effective the new invariants are at detect-

ing execution anomalies. In the second case study, we explore whether inferred invariants can be helpful to distinguish meaningful differences between a testing and a deployed environment. All the invariant inferring tests were conducted on a Macbook Pro laptop, with a 2.5 GHz Intel Core i5 processor, 4 GB 1600 MHz DDR3 memory.

6.1 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 Fig. 1 and has three main components: the UAV (Ascending Technologies Hummingbird), the moving platform (iRobot Create with a mounted landing platform of 50 cm × 50 cm, 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 motion capture room and provide the UAV with the position of the iRobot.

6.1.1 Setup

The training to collect traces process was conducted under what we determined were normal operating conditions. The UAV can takeoff from anywhere in a 8 m × 8 m 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 15 cm of the iRobot's center for 1.5 s. These values were derived empirically under normal operating conditions with a goal of balancing the success rate with the speed of landing. Overall, the success rate for this system is still relatively low (approximately 55 %), but we did not further refine the system to see if the invariant monitoring approach could also detect and avoid such failures under normal conditions.

To generate invariants for the system, we collected trace bags from 83 successful runs from more than 150 total trials. We consider a run successful when the UAV lands on the iRobot, turns off its motors, and remains on the platform for 5 s. 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. The topics $iRobot$ and UAV contained the current position (x, y, z) and attitude (pitch, roll, yaw) for these vehicles and the $task$ topic contained the target position and attitude for the UAV. The fourth topic, $state$, contained state information (e.g., startup, launch, hover, task, land, shutdown) of the controlling system. In the end, the trace files for invariant generation contained over nine million variable-value pairs.

Next, the processed data traces were fed to the extended daikon inference engine. Besides the default invariant templates, we activated the time-related and polygon invariant templates (at the time of the assessment we had not implemented the other templates yet), and run daikon twice to get the condition invariants based on the value of $state$ messages and then invariants for each state partitioned. The inference process took 6 min 20 s 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 (Perkins and Ernst 2004) 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 Sect. 7. With these invariants and the actions defined in the configuration file, the tool synthesized the monitor node and a revised launch file so that the monitor could run 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 to abort a landing. 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 3). These scenarios were developed to test the performance of the system with and without the synthesized monitor under normal conditions (similar to the training set, except the iRobot was

Table 3 Evaluation scenarios

ID	Name	Description	Success criteria
$s1$	Normal	Similar to 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 UAV	Succeeds on landing
$s7$	False airport	iRobot position is incorrect and no vehicle is located there	Succeeds if it avoids landing

Table 4 Summary of results across all scenarios

	Scenario		% Success	Avg. time to land (s)	Sample invariants violated during failures
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	$iRobot_freq \geq 2.04876$
	$S6$	Steal vehicle	20	41.6	$-0.457771 \leq iRobot.x_rate \leq 1.01126$ $-0.532218 \leq iRobot.y_rate \leq 0.962376$
	$S7$	False airport	0	–	$UAV.z \geq 0.245868$
	Scenario		% Success	Avg. time to land (s)	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	–	–

constrained to a smaller 4 m × 6 m area) 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.

6.1.2 Results

For each of the scenarios, we performed 5 trials with and without the invariant monitor, except for the normal scenario where we ran 20 trials. Table 4 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. The system with the monitor worked more safely that it did without the monitor, succeeding with a higher rate across all scenarios.

For the successes, the base system took an average of 35.5 s to succeed, while the system with the monitor took 62.8 s to succeed. The increased time per trial is due to invariant violations while attempting to land that cause the system to abort and try again. Figure 11 shows a box plot depicting the

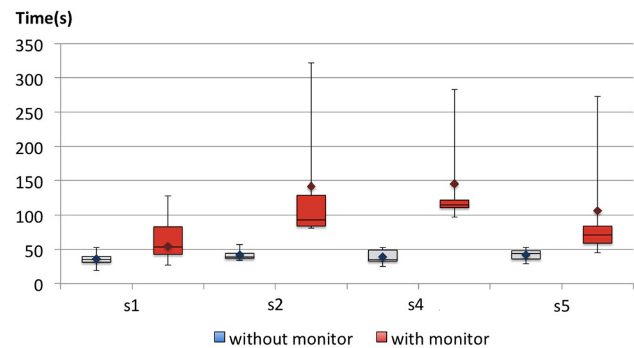


Fig. 11 Time to land

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 caused by landings that are aborted, causing another landing attempt, when invariants are violated. Not all of these invariant violations, however, are linked with imminent failures. The monitor only allows the UAV to land when all the invariants are satisfied. In the best case, this will happen on

the first attempted landing, but in most cases it requires a number of landing attempts within a single trial.

For the approach to be cost-effective, the cost of a failure must be higher than the frequency of violated invariants times the cost of taking a corrective action. In this particular case study, not-crashing the vehicle clearly outweighs delaying the landings, making the approach appealing in this setting. Like argued earlier, inferring invariants at a higher level of granularity and on an activity with limited variability reduces the number of invariant violations, ultimately contributing to the effectiveness of the approach.

In addition, we also analyzed the performance overhead of adding the monitor. On average, the monitor adds a 0.35 ms latency to the published messages. For comparison, moving a ROS node from running locally to another computer on an Ethernet network adds approximately 0.5 ms to the latency. Thus, the overhead of the monitor is less than that of changing the distribution of nodes in a distributed ROS system.

We now look at the details of three of the scenarios ("normal," "wind blowing," and "fragile platform") and examine the invariants and contexts, before briefly discussing the other scenarios ("occupied landing," "slowed link," "stealing vehicle," and "false airport").

Normal scenario: In the normal scenario, most failures were caused by the iRobot's suddenly changing direction while the UAV was trying to land. Figure 12 (top) shows the 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 often 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. We also see that the overall number of failures for the "normal" scenario increased from 45 to 65 % as compared to training due to the smaller operational area of 6 m × 4 m for "normal" versus 8 m × 8 m for training. 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 and $iRobot$ positions, speeds, and rotations during the landing process. Figure 12 (middle) shows the y axis polygon invariant between the UAV and iRobot ($UAV.y + 0.0554{*}iRobot.y \geq -1.89 \cap UAV.y - 0.990{*}iRobot.y \leq 0.151 \cap UAV.y - 1.081{*}iRobot.y \geq -0.202 \cap UAV.y + 1.732{*}iRobot.y \geq -4.664 \cap \ldots$) 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.

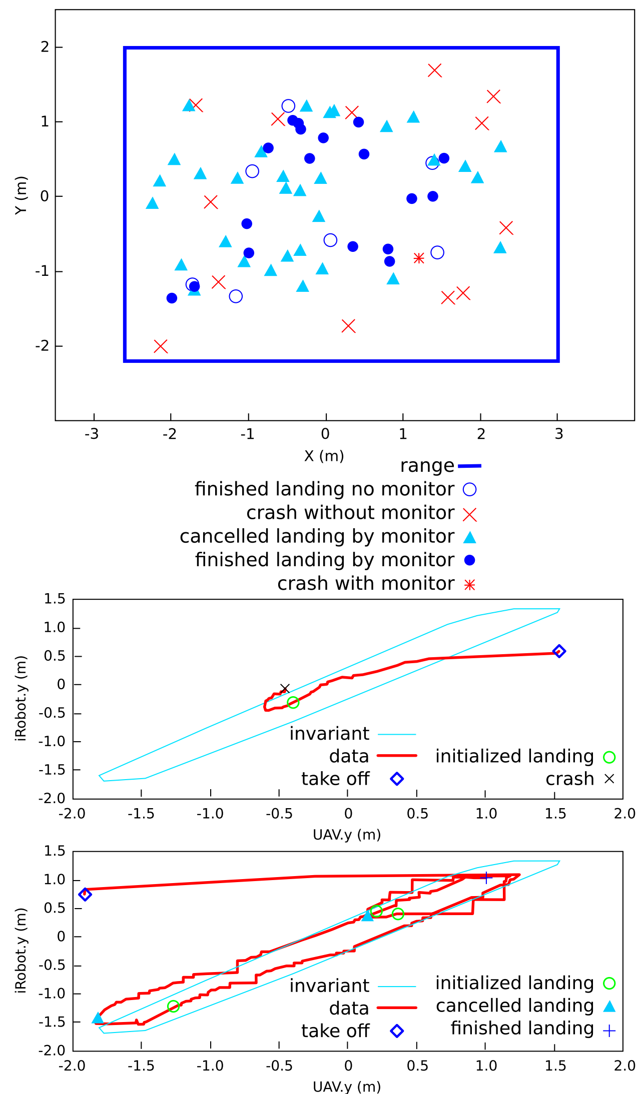


Fig. 12 Normal scenario area (*top*), invariant violation without monitor (*middle*), and with monitor (*bottom*)

In contrast, Fig. 12 (bottom) 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.

Wind blowing scenario: In the wind blowing scenario, the strong wind provided by a fan 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 UAV, as described in Table 4. Figure 13 (top) shows the locations where landings occurred. None of the landings occurred within 2 m 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

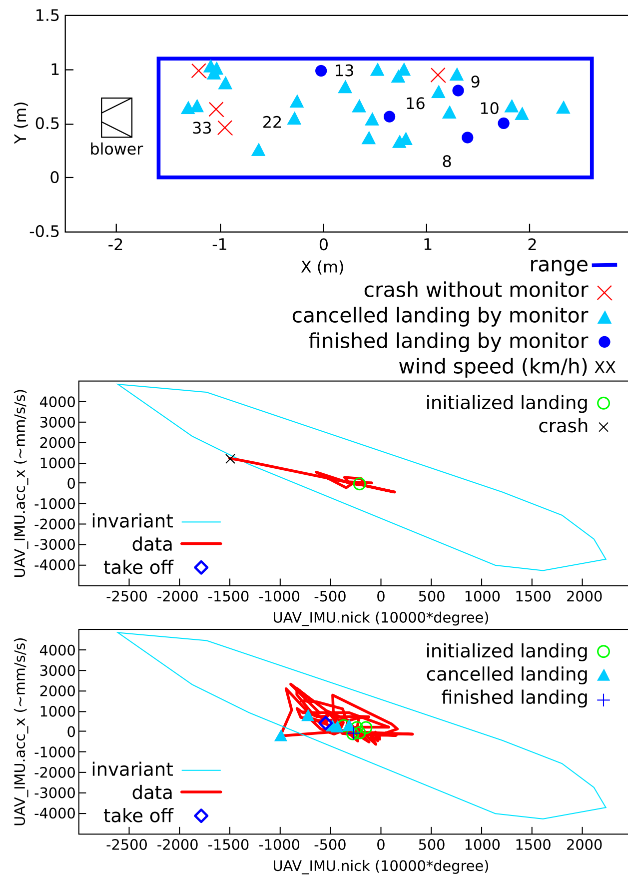


Fig. 13 Wind blowing scenario area (*top*), invariant violation without monitor (*middle*), and with monitor (*bottom*)

successfully land. The system with the monitor was able to detect invariant violations to prevent the landing when it was unsafe and was able to land every time. Figure 13 shows two different trials without the monitor (middle) and with the monitor (bottom) for the polygon invariant. This particular invariant involves the relationship between nick/pitch and acceleration on the x-axis. In Fig. 13 (middle) the UAV leaves the polygon and crashes almost immediately when it attempts to land when it is outside the polygon invariant region, as indicated by an x in the figure. In Fig. 13 (bottom) the violation of the invariant while using the monitor leads to a landing reinitialization, avoiding a crash. The other landing reinitializations in Fig. 13 (bottom) came from the violation of other monitored invariants.

Fragile platform scenario: In the fragile platform scenario (see Fig. 14), the landing platform was broken so that it tilted if the UAV did not land in the upper right quadrant as shown in Fig. 14. 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 15 (middle) 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 15 (bottom) 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 suc-

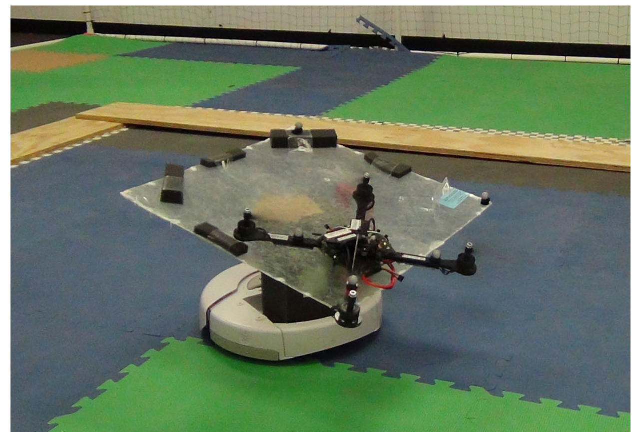


Fig. 14 UAV attempts to land on fragile platform

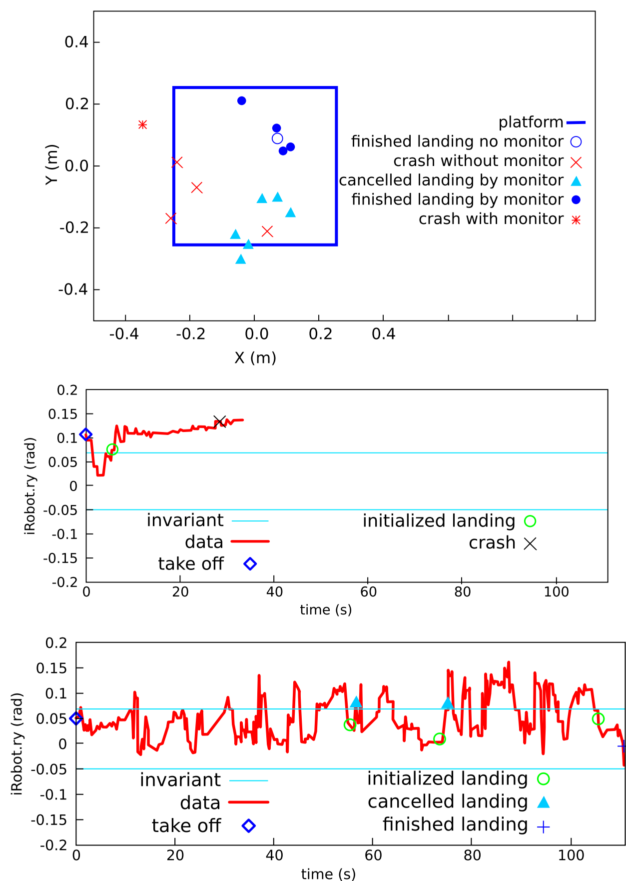


Fig. 15 Fragile platform scenario area (*top*), invariant violation without monitor (*middle*), and with monitor (*bottom*)

cessfully land 80 % of the time, while without the monitor it was only successful 20 % of the time.

Occupied landing scenario: 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 4 as $UAV.z \leq 0.371295$, which means in normal cases the height of the UAV should be lower than 0.371295 to finish its landing sequence. The monitor detected the violation on the invariant, and it canceled the landing, which we consider a success.

Slow link scenario: In the slow link scenario, the message rate from the iRobot position was periodically (every 5 s) slowed down to 0.5 Hz. to mimic a faulty positioning sensor or a radio link that drops packets. The monitor detected this abnormal situation by the invariant $iRobot_freq \geq 2.04876$ on message frequencies as shown in Table 4. 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.

Stealing vehicle scenario: 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 4, the invariant are $-0.457771 \leq iRobot.x_rate \leq 1.01126$ and $-0.532218 \leq iRobot.y_rate \leq 0.962376$, 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, which we considered a success.

False airport scenario: In the false airport scenario there was no iRobot, instead 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.
Summary: Overall, in this case study we can see that:

- The inferred invariants monitor decreases the failure rate when encountering unexpected situations from 76 to 11 %, but its efficiency may suffer;

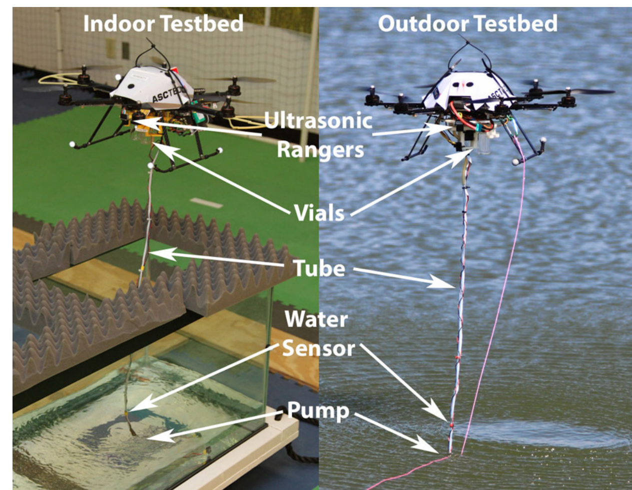


Fig. 16 Indoor and outdoor water sampling (Ore et al. 2015)

- 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.

6.2 Case study 2: water sampling UAV

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 issues associated with changes in the environment. In this case study, we explore that conjecture using a UAV water sampling system where we had access to bags of trace data from test and deployed runs (Ore et al. 2015). The system is designed to autonomously fly over a body of water, approach predefined locations, and sample the water through a pump, as shown in Fig. 16. Since the water pump can only work within 1 meter of the water, the most challenging part of this system is the precise height control over water. To do this, the system uses a combination of ultrasonic sensors, air-pressure altimeter, GPS, and conductivity sensors to estimate the height above water.

6.2.1 Setup

The system was first tested in a controlled environment, as shown in Fig. 16, where the UAV flew over and sampled water from a fish tank. We collected trace bags from 16 successful indoor runs, when the UAV started 2–3 m away from the fish tank, flew over the tank, then descended and sampled water three times, and finally flew back and landed. We identified, with the guidance of the core developer, the most critical topics worth monitoring during the water sampling stage. We selected the default invariant templates, and the new time-related, polygon, architecture, and temporal invari-

ant templates. This effort 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 117 s. We then used the classifier and three additional successful indoor traces to further prune the set of invariants. The resulting pruned set had 533 invariants, which included 229 default, 64 time-related, 233 polygon, 4 architecture, and 3 temporal invariants. The noticeable reduction in the number of polygon invariants was expected given that the number of training runs was not enough to discard many accidental relationships found between pairs of independent variables. For example, our approach inferred an invariant combining the UAV's speed and heading angle even though they are intuitively independent. Still, this was dramatic enough to indicate that the filtering process for this type of invariant needs to be more aggressive. We discarded these invariants from further analysis of this system.

6.2.2 Evaluation

We used a bag from a successful outdoor run in which the system flew about 30 m over the lake of more than 1 km^2, sampled water from it, and checked it against the set of inferred invariants. We found that, excluding the polygon invariants, 276 were shared by the indoor and outdoor environments, while 24 were broken as is shown in Table 5.

Among the violated invariants, a few are worth highlighting because they reflect environmental changes. The frequency invariant on *pose* conveys that localization signals operate at least at 20 hz. That is true when operating in the indoor environment with Vicon support. Outside, however, localization is provided by GPS which operates at a much slower frequency and hence the violation. This change from Vicon to GPS also generates a violation of the architectural invariants, and the ordered-pair temporal invariant as the lower frequency of the GPS signal made the interval between *imupose* greater than expected. The violation of acceleration and attitude invariants are caused by the more aggressive maneuvers performed by the PID controller as the UAV navigates larger distances and fights the wind. Based on this, the water sampling team revised the indoor test environment to better match the outdoor.

Through an analysis of the invariants with the primary developer we were also able to find and explore several erroneous assumptions. For example, the developer was surprised by the lack of an invariant indicating that, when the pump is on, the connectivity sensor should always be wet. To explore this issue we incorporated in the invariant list this supposedly missing invariant, and then used the checker against all the bags to help pinpoint the data that violated this invariant. We provided this finding to the developer, and ended up locating a subtle problem in the on-board pump controller that would erroneously set the pump state. This extraneous pumping while the pump was not in water likely resulted in the early failure of a previous pump.

Table 5 Check result of outdoor testcase

Invariants violated
$sampler_raw.H2O_1 >= 306$
$sampler_raw.H2O_5_variance <= 14.0$
$pose.rotation.x <= 0.0535418$
$pose.rotation.y >= -0.0804302$
$pose.translation.x >= -1.99948$
$pose.translation.y < pose.translation.z$
$pose.translation.y <= 0.32948$
$pose_Freq >= 20.0$
$imu.acc_angle_nick >= -4284$
$imu.acc_angle_roll < imu.mag_z$
$imu.acc_angle_roll <= 2516$
$imu.acc_x_calib >= -747$
$imu.acc_y_calib >= -439$
$imu.angle_nick >= -5099$
$imu.angle_roll <= 3192$
$imu.angvel_nick < imu.mag_z$
$imu.angvel_nick <= 2173$
$imu.height_reference! = 0$
$imu.mag_x <= 772$
$imu.angleYaw_variance >= 5934.0$
$pose : (\{vicon\}, \{vicon\})$
$gps : (\{\}, \{\})$
$imu \rightarrow pose : \{0.0000524, 0.0621, \phi, 2 \times raw + 2 \times imu\}$

Overall, this case study shows that the proposed approach can be useful to check:

- Developers' expectations and assumptions, and help pinpoint the context of the inconsistencies if there is an invariant violation.
- Differences of system behaviors under various environments or deployments.

7 Conclusion and future work

We have introduced a general approach for automated invariant inference and monitoring, and instantiated it in the context of ROS so that robotic systems implemented with this operating system can leverage it with minimal effort. The approach is able to automatically infer specialized invariants for a robotic system based on a training set, and to detect the violation of those invariants to avoid failures under various scenarios. The case studies illustrate the potential of the approach and toolset in failure detection, potential recovery

when facing unexpected situations, and checking for differences between deployed environments.

The case studies also show the approach limitations. An essential limitation is that the inferences are generated based on the observed behavior of the system, and these observations are partial and incomplete. They are partial in that they reflect particular dimensions of the system behavior that we deem worth capturing based on the analysis of their cost-effectiveness. They are incomplete in that the training set is finite, and often small. This means that the inferred invariants may over-approximate the potential system behavior (missing some violations), and they may also under-approximate it (generating false positives). Other limitations are those associated with our current set of inference templates, their refinement, and the implementation of the different stages of the approach.

In the short term, we continue to evolve the inference templates and aim for a broader assessment of the approach in practice. Longer term, we are interested in exploring the application of filters based on the variance and pedigree of a variable as well as the automatic identification of redundant messages. Within the area of monitoring, we would like to further investigate sampling schemes that can reduce overhead while minimizing information loss without relying on the engineer's domain expertise. We would also like to explore richer invariants that can encode the information in multiple message sequences and that can support probabilistic expressions to better capture the uncertainty present in robotic systems. Last, we would like to enrich the actions we support after an invariant is evaluated. Satisfied invariants could help refine the space of programmable actions available to the user. Alternatively, violated invariants could be rectified so that minimally reformulated messages that remain within the system invariants are guaranteed to be published.

Acknowledgements This work was partially supported by Air Force Office of Scientific Research #FA9550-10-1-0406, United States Department of Agriculture National Institute of Food and Agriculture #2013-67021-20947, and National Science Foundation CSR-1217400. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., & Tang, A. (2015). Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Transactions on Software Engineering*, *41*(7), 620–638.

Barham, P., Donnelly, A., Isaacs, R. & Mortier, R. (2004). Using magpie for request extraction and workload modelling. In: *OSDI'04 Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation* (pp. 259–272).

Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M. & Ernst, M. D. (2011). Leveraging existing instrumentation to automatically infer invariant-constrained models. In: *ESEC/FSE '11, Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (pp. 267–277). New York, NY: ACM.

Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A. & Brewer, E. (2004). Path-based failure and evolution management. In: *Proceeding NSDI'04 Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (pp. 309–322).

Csallner, C., Tillmann, N. & Smaragdakis, Y. (2008). Dysy: Dynamic symbolic execution for invariant inference. In: *ICSE* (pp. 281–290).

Dwyer, M. B., Avrunin, G. S. & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In: *ICSE '99, Proceedings of the 21st International Conference on Software Engineering* (pp. 411–420). New York, NY: ACM.

Eddy, W. F. (1977). A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software (TOMS)*, *3*(4), 398–403.

Ernst, M. D., Cockrell, J., Griswold, W. G. & Notkin, D. (1999). Dynamically discovering likely program invariants to support program evolution. In: *ICSE* (pp. 213–224).

Ernst, M. D., Perkins, J. H., Guo, P. J., Mccamant, S., Pacheco, C., Tschantz, M. S., et al. (2006). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, *69*(1), 35–45.

Gabel, M. & Su, Z. (2008). Javert: fully automatic mining of general temporal properties from dynamic traces. In: *FSE* (pp. 339–349).

Gabel, M. & Su, Z. (2010). Online inference and enforcement of temporal properties. In: *ICSE '10, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (pp. 15–254). New York, NY: ACM.

Ghezzi, C. & Kemmerer, R. (1991). Astral: An assertion language for specifying realtime systems. In A. van Lamsweerde & A. Fugetta (Eds.), Lecture Notes in Computer Science, *ESEC '91* (Vol. 550, pp. 122–146). Berlin: Springer.

Gillula, J. H. & Tomlin, C. J. (2012). Guaranteed safe online learning via reachability: Tracking a ground target using a quadrotor. In: *2012 IEEE International Conference on Robotics and Automation (ICRA)*.

Golombek, R., Wrede, S., Hanheide, M. & Heckmann, M. (2010). Learning a probabilistic self-awareness model for robotic systems. In: *IROS* (pp. 2745–2750).

Golombek, R., Wrede, S., Hanheide, M. & Heckmann, M. (2011). Online data-driven fault detection for robotic systems. In *IROS* (pp. 3011–3016).

Grunske, L. (2008). Specification patterns for probabilistic quality properties. In: *ICSE '08, Proceedings of the 30th International Conference on Software Engineering* (pp. 31–40). New York, NY: ACM.

Hangal, S. & Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection. In: *ICSE* (pp. 291–301).

Henkel, J., & Diwan, A. (2003). Discovering algebraic specifications from java classes. In *ECCOP* (pp. 431–456). Springer.

Jiang, H., Elbaum, S. G., & Detweiler, C. (2013). Reducing failure rates of robotic systems though inferred invariants monitoring. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 1899–1906). Tokyo, November 3–7, 2013.

Konrad, S. & Cheng, B. (2005). Real-time specification patterns. In *ICSE 2005. Proceedings of 27th International Conference on Software Engineering* (pp. 372–381).

Li, J., Xu, A. & Dudek, G. (2011). Graphical state space programming: A visual programming paradigm for robot task specification. In *ICRA 2011 IEEE International Conference on Robotics and Automation* (pp. 4846–4853). Shanghai, May 9–13, 2011.

Lorenzoli, D., Mariani, L. & Pezzé, M. (2008). Automatic generation of software behavioral models. In *ICSE '08, Proceedings of the 30th International Conference on Software Engineering* (pp. 501–510).

Mendoza, J., Veloso, M. & Simmons, R. (2012). Motion interference detection in mobile robots. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 370–375).

Muller, J. & Sukhatme, G. (2014). Risk-aware trajectory generation with application to safe quadrotor landing. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014)* (pp. 3642–3648).

Ore, J. P., Elbaum, S., Burgin, A., & Detweiler, C. (2015). Autonomous aerial water sampling. *Journal of Field Robotics*, *32*(8), 1095–1113.

Perkins, J. H. & Ernst, M. D. (2004). Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering* (pp. 23–32).

Pettersson, O. (2005). Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, *53*, 73–88.

Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A. & Vahdat, A. (2006). Pip: Detecting the unexpected in distributed systems. In *NSDI'06 Proceedings of the 3rd conference on Networked Systems Design and Implementation* (pp. 115–128).

Sagdeo, P., Athavale, V., Kowshik, S. & Vasudevan, S. (2011). Precis: Inferring invariants using program path guided clustering. In *ASE '11, 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 532–535).

Sattar, J. & Dudek, G. (2014). Reducing uncertainty in human-robot interaction: A cost analysis approach. In *Experimental Robotics* (pp. 81–95). Springer.

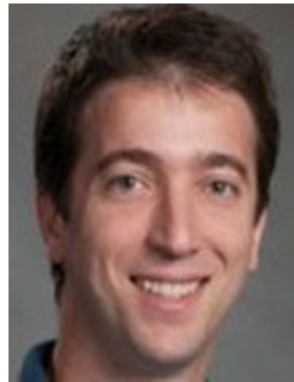
Sattar, J., Xu, A., Dudek, G. & Charette, G. (2010). Graphical state-space programmability as a natural interface for robotic control. In *ICRA 2010, IEEE International Conference on Robotics and Automation* (pp. 4609–4614). Anchorage, AK, May 3–7, 2010.

Steinbauer, G., Morth, M. & Wotawa, F. (2005). Real-time diagnosis and repair of faults of robot control software. In *RoboCup* (pp. 13–23).

Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques* (2nd ed.). Burlington: Morgan Kaufmann.

Yang, J., Evans, D., Bhardwaj, D., Bhat, T. & Das, M. (2006). Perracotta: Mining temporal API rules from imperfect traces. In *ICSE* (pp. 282–291).

Hengle Jiang is a software engineer at LI-COR Biosciences, Lincoln, Nebraska. He received his B.S. in 2002 from Qingdao University and M.S. in 2014 from Computer Science and Engineering Department at the University of Nebraska-Lincoln, where he was a research assistant in Nebraska Intelligent MoBile Unmanned Systems (NIMBUS) Lab. His research interests include software testing and analysis for small aerial robots.



Sebastian Elbaum is a Professor in the Computer Science and Engineering Department at the University of Nebraska - Lincoln. His research aims to improve system dependability through testing, monitoring, and analysis. He is the recipient of an NSF Career Award, IBM Innovation Award, Google Faculty Research Award, and 4 ACM SigSoft Distinguished Paper Awards. He served as Program Chair for the ISSTA 2007 and ESEM 2008, and as Co-Editor for the Information and Software Technology Journal. He is currently on the Editorial Board of the ACM Transactions on Software Engineering and Methodologies Journal and is the program co-chair for the 2015 International Conference on Software Engineering. He is a co-founder of the EUSES Consortium, the E2 Software Engineering Group at UNL, and the Nimbus UAV Lab at UNL. He received his Ph.D. from the University of Idaho, and a Systems Engineering degree from Universidad Catolica de Cordoba, Argentina.

Carrick Detweiler is an Assistant Professor in the Computer Science and Engineering department at the University of Nebraska - Lincoln. He co-directs and co-founded the Nebraska Intelligent MoBile Unmanned Systems (NIMBUS) Lab at UNL, which focuses on developing software and systems for small aerial robots and sensor systems. Carrick obtained his B.A. in 2004 from Middlebury College and his Ph.D. in 2010 from MIT CSAIL. He is a Faculty Fellow at the Robert B. Daugherty Water for Food Institute at UNL and recently received the 2014 College of Engineering Henry Y. Kleinkauf Family Distinguished New Faculty Teaching Award. He is currently lead PI on NSF and USDA grants, including a National Robotics Initiative Grant. In addition to research activities, Carrick actively promotes the use of robotics in the arts through workshops and collaborations with the international dance companies Pilobolus and STREB.