Reducing Failure Rates of Robotic Systems though Inferred Invariants Monitoring

Hengle Jiang, Sebastian Elbaum and Carrick Detweiler

***Abstract*— System monitoring can help to detect abnormalities and avoid failures. Crafting monitors for today's robotic systems, however, can be very difficult due to the systems' inherent complexity. In this work we address this challenge through an approach that automatically infers system invariants and synthesizes those invariants into monitors. The approach is novel in that it derives invariants by observing the messages passed between system nodes and the invariants types are tailored to match the spatial, temporal, and operational attributes of robotic systems. Further, the generated monitor can be seamlessly integrated into systems built on top of publish-subscribe architectures. An application of the technique on a system consisting of a unmanned aerial vehicle (UAV) landing on a moving platform shows that it can significantly reduce the number of crashes in unexpected landing scenarios.**

I. Introduction

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

Consider, for example, the scenario illustrated in Figure 1 where a small unmanned aerial vehicle (UAV) is autonomously following and attempting to land on a moving platform whose location is continuously fed to the UAV. A typical landing test consists of the UAV starting a few meters away from the platform, finding and following the moving platform, and then initiating the landing sequence. Using a standard message passing system such as ROS (Robot Operating System) [5], this system contains over thirty nodes that communicate through dozens of message channels.

An engineer developing a monitor to detect anomalies for this kind of system is likely to focus on a small subset of the variables and relationships between variables. For example, a monitor crafted for this system would likely check whether the positions of the UAV and the platform are aligned when landing is initiated, and the speed of the platform is less than a safe maximum. There are, however, many other aspects of the system worth monitoring that are more subtle and may not be considered by the engineer given the number of variables and relationships involved. For example, it may help to ensure that the platform is horizontal and not rotating when landing, the UAV's angles are not greater than a multiple of the UAV's commanded velocity, there is only one landing platform reporting its location, and the platform is unoccupied and able to support the weight of the UAV.

All authors are with the NIMBUS Lab in the Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE 68588, USA. {hjiang, elbaum, carrick}@cse.unl.edu
This work was partially supported by NSF CNS-#1217400 and AFOSR #FA9550-10-1-0406.

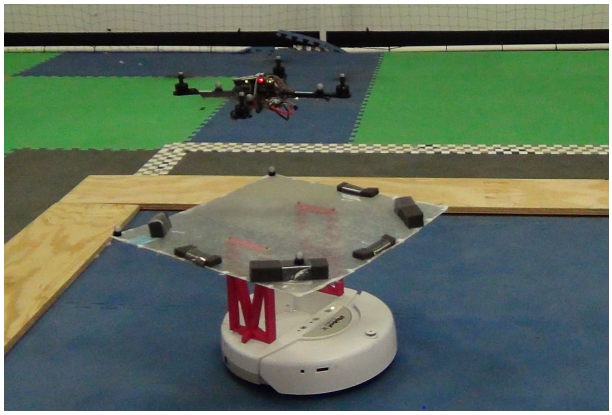

Fig. 1: UAV attempting to land on moving platform.

It is unlikely that the system engineer will consider all possible variables and relationships. To alleviate this challenge, we propose an approach to automate the synthesis of monitors from the traces of robotic systems. The approach is inspired by existing software engineering approaches for automated invariant inference [7]. 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 template invariants utilizing the trace values, and dropping or refining the ones that are falsified by other trace values. For example, given a template invariant $varX \geq constant$ and a trace of six variable-value pairs collected from time $t1$ to time $t6$, $tr = \{t1 : a = 1, t2 : b = 3, t3 : a = 1, t4 : a = 2, t5 : a = 1, t6 : a = -1\}$, the approach would instantiate the invariant template as $a \geq 1$ after reading the value of a at $t1$ and further support it until $t6$ when value $a = -1$ is observed and it becomes necessary to refine the invariant to $a \geq -1$; for variable b an invariant may not be reported as there may not be enough values to support that instantiation. Given a set of traces, the inferred invariants provide a characterization of the behavior of the system as exhibited in those traces.

Existing techniques to automatically infer invariants have been shown useful for generating generic invariants like the one illustrated above to act primarily as a function's pre and post conditions. The application of these techniques to large distributed robotic systems, however, has been limited. We conjecture that this is due to the focus on the generation of low level invariants which is impractical in these large systems, the lack of domain-specific invariants that capture

the temporal and spatial aspects of robotic systems, and the lack of tools to seamlessly integrate such approaches into the development process and common toolsets.

Through this work we aim to make automated invariant inference techniques amenable to robotic systems. First, we operate at the granularity of messages commonly used by robotic systems operating under a publish and subscribe architecture. This reduces the overhead typically suffered by similar approaches and it lets us infer properties related not just to program states but also to message sequences, which are critical to most robotic systems. Second, we have developed invariant templates that account for properties that are deemed important in the context of robotic systems such as those characterizing the relationship between variables that have a continuous distribution such as sensors values, those including a temporal component to capture the derivatives of raw variable values, and those that can differentiate among system operating modes. Third, we have implemented a version of the approach that automatically synthesizes a monitor as a node that can be seamlessly integrated into existing ROS system. The monitor can be tailored to trigger actions or send messages when an invariant is violated.

In the context of the previous scenario, our approach automatically synthesized a monitor that checks 1059 invariants over 56 system message variables, and includes the subtle ones we mentioned earlier. The study illustrates the approach's potential to increase the system robustness in the presence of new landing scenarios by more than a threefold by preventing landing under conditions that violate an inferred invariant. The contributions of this work are:

- Extension of automated induction techniques that derive properties for robotic systems from execution traces
- Design and implementation of a ROS toolset that can derive system properties and synthesize a monitor to detect property violations and initiate recovery actions on *any* ROS system with minimal user effort
- A case study illustrating the potential of the techniques and toolset in reducing the failure probability of a UAV landing on a moving platform.

II. Approach

A. Overview

The goal of our approach is to enable the automatic generation of system monitors that can detect anomalous behavior and launch counter-measures. The type of system we target is a robotic system made of distributed nodes that sense and actuate, and that communicate through some form of message passing scheme. As mentioned, our work was motivated and implemented in the context of ROS, but the approach is generalizable to other similar message passing infrastructures (e.g., LCM [3], Microsoft Robotics Developer Studio [4] , CLARAty [1]).

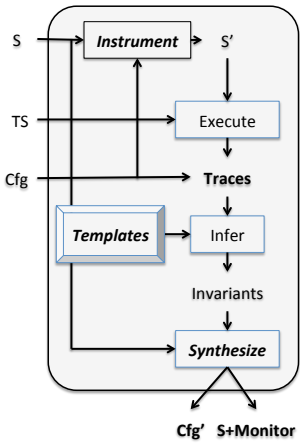

Fig. 2: Approach Overview.

Figure 2 provides an overview of the approach, which is similar to what is currently performed by existing dynamic invariant inference frameworks [7]–[9], [13]; we have highlighted the differences by bolding certain components' labels.

System S and an optional configuration file Cfg serve as the only inputs to the approach. S is instrumented to capture the messages passed between the nodes in the system, constituting system S'. When S' is executed with the training set TS, a set of $|TS|$ traces $Traces$ is generated, where each trace will contain a sequence of variable-value pairs found in the messages. The approach will then attempt to instantiate the predefined invariant templates based on the information found in the traces and in the configuration files. Each instantiated invariant is a boolean expression that characterizes the variables values observed in the $Traces$. Last, the invariants generated are synthesized into a monitor that can be incorporated as a node into the system S. Our main contributions are in: the instrumentation to capture messages passed between system nodes; the inclusion of information from configuration deployment files; a class of invariant templates that better fit robotic systems; and in the build process to incorporate the invariants into the target system as a monitoring component. We now describe some of these phases in more detail.

B. Richer Sources of Information

Our approach targets two unique sources of information commonly used in robotic systems. First, it targets the messages being passed between the system components. Although traditional dynamic invariant inference approaches commonly capture variable-value pairs at the entry and exit of functions, such an approach would not scale to large robotic systems. Instead, we shifted our focus to the structured messages that are sent between the nodes of robotic systems. We observed that these higher level messages cause less overhead while still providing a rich enough data set from which to generate invariants on a per-node level.

Figure 3 shows a subset of the small UAV and platform system described before where each circle represents a node, and each line represents a channel or topic where messages are published. So, for example, given nodes $/a/ctrl\_state\_machine$ and $/a/car\_ctrl$, a message between those nodes on topic $/a/subject\_ctrl\_state$, may consist of $timestamp$ $123287 : \{state : 8, user\_data : 0\}$. Note that a node may consume and publish messages on different topics. Furthermore, many messages are published during an execution. For our system, in a typical execution scenario, there are hundreds of thousands of messages published.

Now given a trace of messages, we cluster the messages consumed and published by each node. Given such cluster, we pair the messages published by a node, with the messages previously consumed by a node. The idea is that the entry values in the messages consumed by a node are likely to define its behavior and affect its outputs

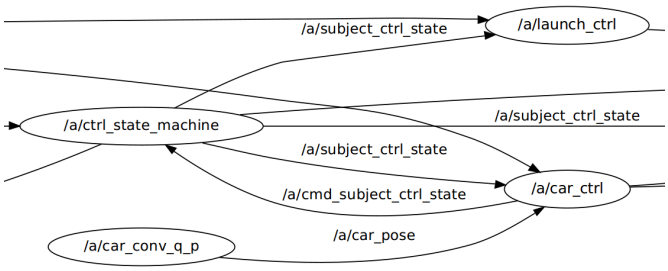

Fig. 3: Message flow across system nodes.

as evident in the published messages. Figure 4 presents a partial message trace at the top (timestamps have been dropped to save space), and the pairings at the bottom for node $/a/car\_ctrl$. In each pairing, a published message on topic $/a/cmd\_subject\_ctrl\_state$ is paired with the last values consumed by the node through the several topics including $/a/subject\_ctrl\_state$ and $/a/car\_pose$. Note that not all messages are published at the same rate, so each pairing includes the published message with the latest value available for all the incoming messages. The approach can be parameterized to relate published values to a range of previously consumed values, increasing precision but also inference cost in the next stage. In the end, these pairing along with their time markings are going to be the key input to the inference engine to identify relationships between the messages incoming to and outgoing from a node.

The second unique source of information that our approach can leverage are the deployment configuration files (Cfg in Figure 2, launch files in ROS) often available in these sys-

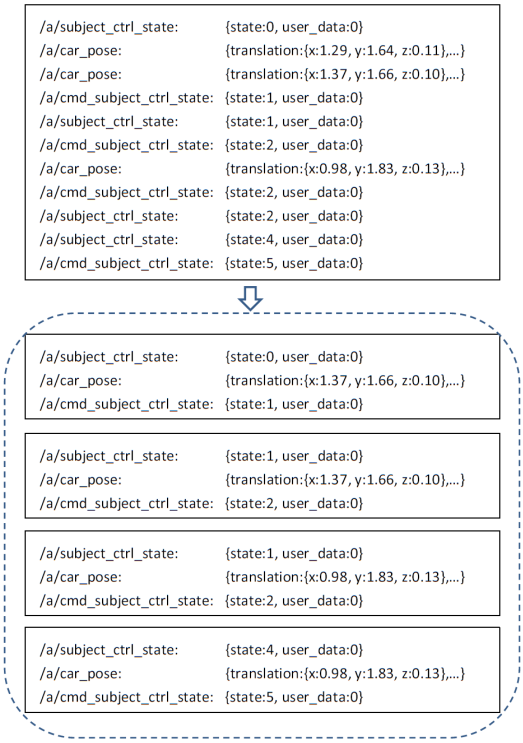

Fig. 4: Sample trace (top) and message pairings (bottom).

tems. These files allow system operators to better configure their systems to match a deployment context. For example, our sample system can operate multiple UAVs simultaneously and change the source of location information by simply tweaking configuration parameters without changing or rebuilding the source code. The configuration parameters also serve as input to the inference engine since the messages published by nodes may be conditioned by them.

C. Invariant Template Family

Techniques that infer invariants from program execution often target a set of standard invariants such as the memory locations read or written at marked program points [13] or the ranges of values observed for a variable at the entry or exit points of a function [2]. Identifying richer invariants, like the ones we aim to capture in robotic systems, requires the specification of richer invariant templates. Through this work we introduce three new types of invariant templates that reflect the spatial and temporal nature of robotics systems.

First, we introduce invariant templates that incorporate time as a central component of invariants. The simplest of these templates serves to characterize the messages average frequency and variance. For average, this may takes the form of $constantLower < \sum_{i=1}^{n}(msgtime_i - msgtime_{i-1})/n < constantUpper$. For our sample system this type of invariant is useful to detect, for example, stale location data that may direct the UAV to the wrong location. A more complex invariant template involving time 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 takes the form of $constantLower < dVarX/dt < constantUpper$ where $VarX$ is a vector of variable values. In our scenario an invariant of this type is $dLocationUAV = d[loc_i, loc_{i+1}, ..., loc_n]/dt < maxUAVVelocity$. To infer such invariants, the traces are enriched with timing data that are specially labeled so that the inference engine can identify them and associate them with all the other potential variables.

Second, we introduce invariant templates that define relationships between two variables[1] that can be characterize through a polygon. This invariant template takes the form of $\bigcup_i^n (a_iX + b_iY + c[>= \mid <=]0)$ that defines a polygon of n sides. Every time a new variable-value is read, it is checked against the polygon. If it resides inside the polygon, it is ignored. If it resides outside the polygon, the polygon is relaxed by computing the convex hull that includes the new observation. This type of invariants are valuable to capture physical space bounds. For example, if our system operating scenario was bounded by the dimensions of a room and an attached hallway, this invariant template would be instantiated with at most eight sides. This type of invariant can also characterize relationships between variables that are hard to anticipate because their lack of linearity. Take the

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

UAV acceleration and its pitch and roll. Ideally, these 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.

Third, we introduce the notion of conditional state invariants, that is, invariants that can only hold under certain system states that can be identified as such. For example, in our system the invariants that hold when the UAV is on the ground versus when it is flying are quite different. This partition of the space of system behavior helps to generate more and more precise invariants, and it also helps to make the inference process more efficient. Attempting to produce invariants without differentiating such states would result in a smaller set of more general invariants, but it would miss many valuable invariants that only apply to one system state. For example, the critical invariants that characterize how the system should behave when attempting to land on the moving platform (e.g. the UAV and platform X and Y coordinates should be within a certain threshold) would be dropped as they would not hold when the UAV is pursuing the landing platform. To infer such invariants, we use the composition of existing invariants templates. First we identify variables that have a small discrete set of values; this helps us identify variables such as $UAVmodes$ which has a range from 1 to 9 indicating whether the UAV is taking off, hovering, translating, landing, etc. Second, we partition the traces into sub-traces according to those discrete values. Third, we perform inference on the sub-traces independently and incorporate the learned invariants and a predicate on the discrete variable as part of the monitor.

At the end of the inference process, invariants that are statically justified (as per the number of observations and their variance) are reported.

D. Monitor Synthesis and Implementation

Given a set of generated $Invariants$ consisting of boolean expressions over messages of type $Topic$, the synthesis process consists of the creation of a node that consumes messages of type $Topic$ and checks whether those messages violate any of the $Invariants$. The monitor also encodes what actions to take if an invariant is violated. The default action is to raise a warning, but the monitor can also be configured to drop messages (in this case the messages are not just consumed by the monitor but intercepted and then re-published if they do not violate an invariant) or generate new messages. In the case of our sample system, the monitor re-launches the UAV landing sequence when an invariant is violated. We note that the synthesis produces a monitor that can work directly with the original system S. The difference resides in the configuration file Cfg' which remaps and extends how messages are passed and incorporates the monitor to process those messages.

We have implemented our approach to work with the Robotic Operating System (ROS). We leverage ROS' capabilities to capture traces in the form of "rosbags" that we post-process to cluster messages and add the necessary information to compute the new invariants. These traces

```
−<monitor inv="iRobot.inv" launch="iRobot.launch">
  −<topic name="/a/state" conditionalInvariantDetect="on">
      <action topic="/a/cmd" value=".state=8"/>
    </topic>
  −<invDetect>
      <bags folder="bags" launch="iRobot.launch"/>
    </invDetect>
  </monitor>
```

Fig. 5: Sample moncfg file to configure the approach

are then feed into the invariant inference engine. The only required user input is a monitor configuration file which follows a traditional launch file structure, and is used to specify the target system and parameters. With that, our approach can infer the invariants, synthesize the monitor, and build a new configuration file CFG' to use the system with the monitor incorporated. The monitor configuration file can specify how to constrain the nodes or messages for which invariants are generated, and to specify the actions to take in case an invariant is violated. Figure 5 shows a sample monitor configuration file where the target is the topic $/a/state$, the traces will be located in $bags$, the launch file is $iRobot.launch$, and the monitor will publish on topic $/a/cmd$ when any coming messages violate any invariants. Our inference process is built on top of the Daikon [2], [8] inference engine. We incorporated the invariants described in Section II-C by post processing the collected system traces and by modifying the Daikon engine. We also built a post-processor of the Daikon output so that it can be synthesized into a monitor.

III. Assessment

The invariant monitoring tools and techniques developed in this paper are general purpose and can immediately be applied to any ROS system. To test our approach, we implemented it on a system designed to land a UAV on a moving platform. The target system was introduced in Figure 1 and has three main components: the UAV (Ascending Technologies Hummingbird), the moving platform (iRobot iCreate with a mounted landing platform of 50cm x 50cm, following its standard "vacuum" motion pattern), and a control system that tracks the iRobot and directs the UAV in its pursuit. For ease of evaluation we run the UAV and iCreate in a Vicon motion capture room and provide the UAV with the position of the iCreate, although it could work with a visual servoing system as well.

A. Training and Evaluation

The training process was conducted under what we determined were normal operating conditions. The UAV can takeoff from anywhere in the 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 it us within 0.15 meters of it for 1.5 seconds. For each landing, the system generated a bag file containing a record of generated messages. To train the system, we collected 83 successful runs. We consider a run successful when the UAV lands on the iRobot, turns off its motors, and remains on the platform for 5 seconds. We collected this number of runs to mitigate the risk of capturing coincidental invariants.

Among all the messages in the collected bags, we chose those published on four topics containing a total of 56 variables for invariant detection and monitoring. Three topics contained position and attitude information: $iRobot$, UAV and $task$ (all $doubles$). The fourth message had the state information of the controlling system: $state$ (unsigned int). Our tool processes the bag files, clustering the messages around nodes, instantiating additional variable-value pairs obtained from the launch files or required by the richer invariant types (e.g., system modes, time stamps), and packaging the traces as required by Daikon. In the end, the trace file for training contains over nine million variable-value pairs.

Next, the processed data traces are fed to the extended Daikon inference engine for analysis. The inference process took 6 minutes 20 seconds to generate 1059 invariants from these traces (this process is known to be polynomial with respect to the number of variables [14] 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 Section V). With these invariants and the actions defined in the monitor configuration file, the tool generates the monitor node and a revised launch file so that the monitor can run alongside the original ROS system without the need for recompilation.

We evaluated the effectiveness of the invariant monitor on 7 different scenarios (shown in Table I). These scenarios were developed to test the performance of the system with and without the monitor under normal conditions (similar to the training set) and under stress. The stress testing scenarios contain unexpected events that the system developer may not anticipate, but that the monitor is able to detect. For the "s3 occupied landing" and the "s7 false airport" scenarios we consider landing as failure and canceling landing as success, while in the other scenarios we set the same criteria for success as set for the training process.

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

TABLE I: Evaluation Scenarios.

B. Results

For each of the scenarios we performed 5 trials with and without the monitor. Table II summarizes the results. Over all the test scenarios, the base system without the monitor succeeded 23.8% of the time, while with the monitor it succeeded 89.4% of the time. Figure 6 plots the success rates for each scenario. It is clear that the system with monitor worked much more safely that it did without monitor, as it succeeded with a much higher rate for all the scenarios.

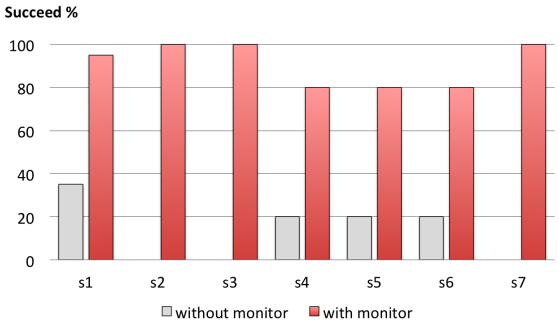

Fig. 6: Landing success rate.

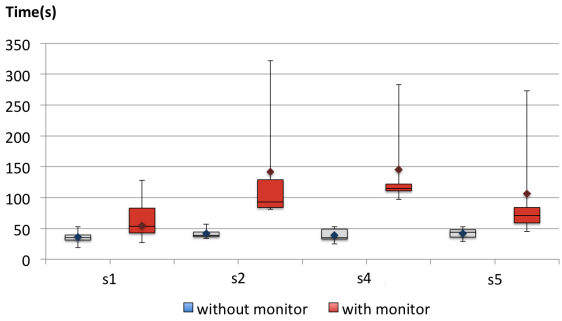

Fig. 7: Time to land (successful scenarios without a monitor).

For the successes, the base "Normal" system took an average of 35.5 seconds to succeed, while the system with the monitor took 62.8 seconds to succeed. Figure 7 shows a box plot depicting the average times with and without the monitor and the variance in these measurements (only for the scenarios in which the system without monitor successfully landed). Without the monitor, the average time has low variance within each scenario and over all scenarios. With the monitor there is a high variance in the time to success. This is because the monitor tends to be conservative. It only allows the UAV to land when all the synthesized invariants are satisfied. In the best case this will happen on the first attempted landing, but in most cases it requires a number of attempts. Also, to monitor the invariants the monitor adds, on average, a 35ms latency to the published messages.

C. Detailed Analysis

We now look at the details for each of the scenarios. Because of space constraints we first describe the "normal", "wind blowing", and "fragile platform" in more detail since they let us introduce different types of invariants and contexts, and then briefly discuss the other scenarios.

In the "normal" scenario, most failures were caused by the iRobot's suddenly changing direction while the UAV was trying to land. Figure 8 shows the successful and failed landings with and without the monitor in the test area where the iRobot was operating. The thicker-blue walls indicate the boundary of the area. The iRobot will typically drastically change directions when it hits a wall, although it occasionally chooses to follow the wall. That is why most of the crashes without monitor are located towards the borders. The single

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

TABLE II: Summary of results across all scenarios.

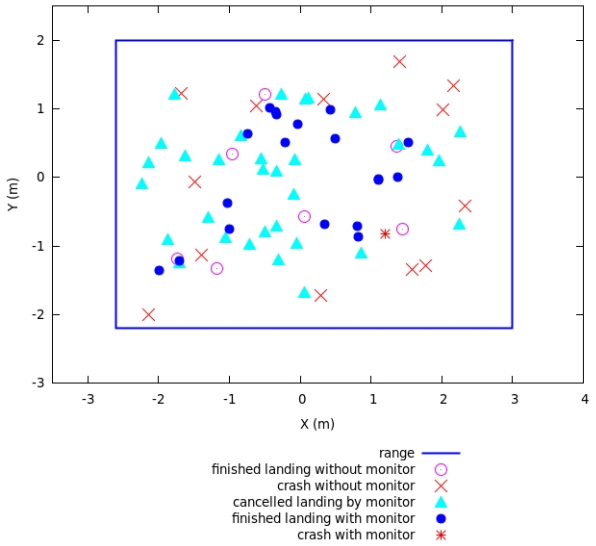

Fig. 8: Outcomes under normal scenario.

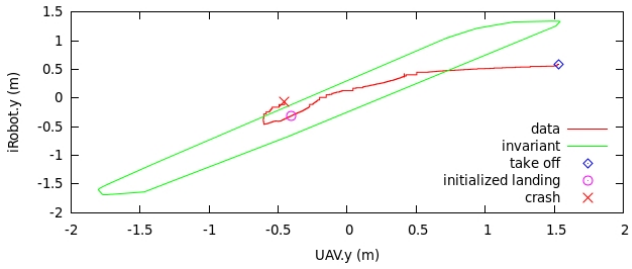

Fig. 9: Normal scenario without monitor.

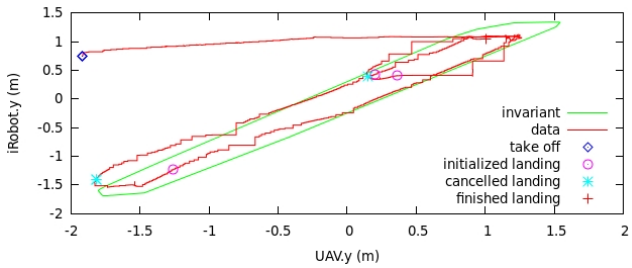

Fig. 10: Normal scenario with monitor.

failure with monitor occurred as the UAV landed on the platform but slid off it because of its incoming speed. When the iRobot quickly changes direction, the monitor detects violations of one of the inferred polygon invariants which characterize the relations between UAV and $iRobot$ positions, speeds, and rotations during the landing process. Figure 9 shows the y axis polygon invariant between the UAV and iRobot without the monitor running. Initially, when the UAV takes off it is outside of this constraint. It then moves over the iRobot and initiates landing. As seen in the figure, it moves out of the polygon invariant while still trying to land and crashes. In contrast, Figure 10 shows the same scenario with the UAV and iRobot position with the monitor enabled. In this case, whenever the constraints are violated the landing is aborted. Eventually, the UAV is able to successfully land while staying within these constraints.

In the "wind blowing" scenario, the strong wind breaks many invariants derived from the normal setup. Neither the system, nor the monitor were designed to explicitly consider wind. However, the monitor is able to detect violations of the UAV and iRobot positions and the roll and acceleration of the vehicle, as described in Table II. Figure 11 shows the locations where landings occurred. None of the landings occurred within 2 meters of the blower where the wind speed was upwards of 33 KPH, which prevented the landing sequence. Even away from the fan, the system without the monitor was unable to successfully land. The system with the monitor was able to detect constraint violations to prevent landing when it was unsafe and was able to land 100%. Figures 12 and 13 show two of the trials with and without monitor for the polygon invariant involving the UAV pitch and acceleration on the x-axis. In Figure 12 the UAV leaves the polygon and crashes almost immediately. In Figure 13, however, the violation of the invariant while using the monitor leads to a landing reinitialization, avoiding a crash (other monitored invariants were violated within the polygon leading to other landing reinitialization as well).

In the "fragile platform" scenario (see Figure 14), the landing platform would tilt if the UAV did not land in the upper right quadrant as shown in Figure 14. The monitor detected the error when checking the violation of

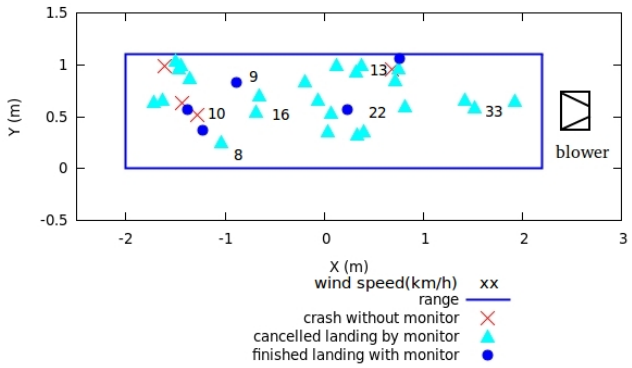

Fig. 11: Outcome under wind blowing scenario.

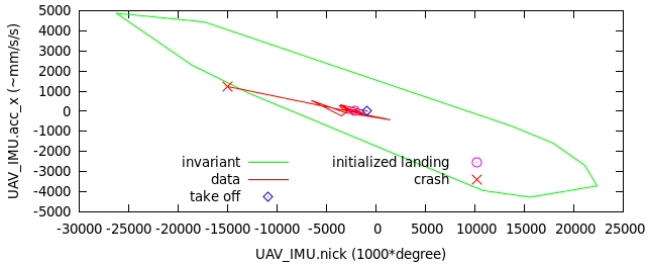

Fig. 12: Wind blowing scenario without monitor.

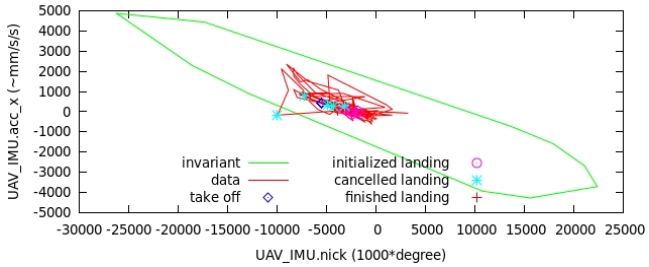

Fig. 13: Wind blowing scenario with monitor.

the invariants on iRobot.rx and iRobot.ry which indicate the horizontal angle of the platform. Figure 16 shows one of the angles without the monitor. The straight lines indicate the bounding constraint inferred by the platform. As shown by the red line, the UAV started to land on the platform, but then the platform tilted and the UAV fell off and crashed. Figure 17 shows the same setup with the monitor. In this case, the UAV initialized landings three times, but in the first two the landing was canceled when the constraints were violated. Overall with the monitor the UAV was able to successfully land 80% of the time, while without the monitor it was only successful 20% of the time.

In the "occupied landing" scenario, the monitor detected that the platform was occupied since it could not decrease its height as it did in the normal case. If it detected this, it canceled the landing, which we consider a success. In the "slow link" scenario, the message rate from the iRobot position was periodically slowed down to 0.5Hz. to mimic a fault positioning sensor or a radio link that drops packets. The monitor detected this abnormal situation by the invariant on the interval between messages and interrupted the landing when the link was not reliable to avoid crashes. In the

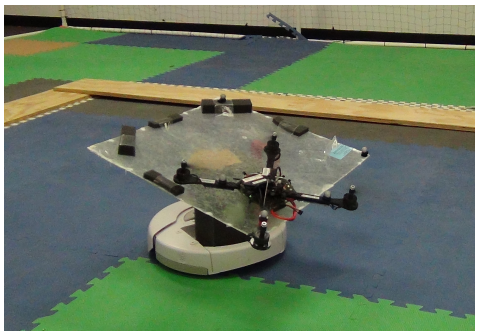

Fig. 14: UAV attempts to land on fragile platform.

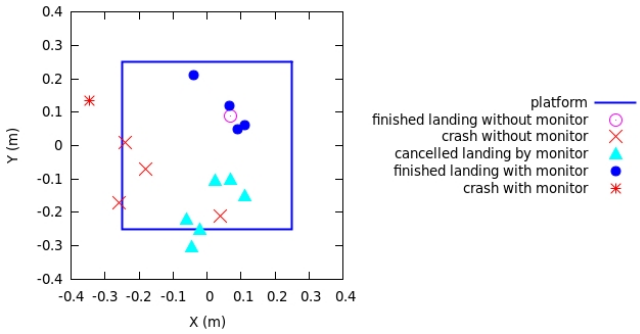

Fig. 15: Outcome under fragile platform scenario.

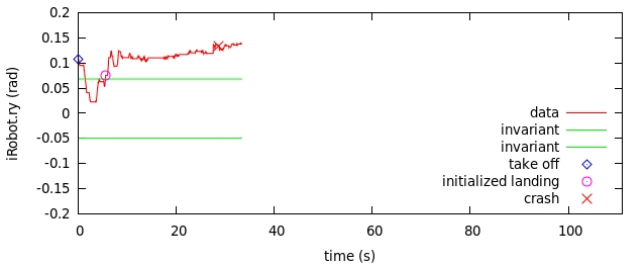

Fig. 16: Fragile platform scenario without monitor.

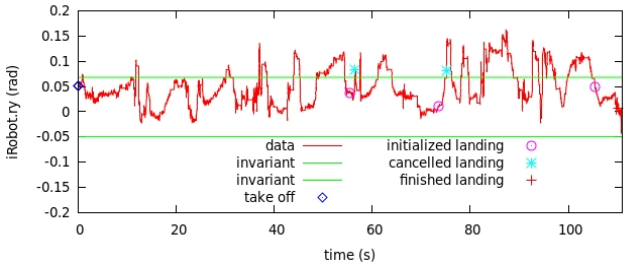

Fig. 17: Fragile platform scenario with monitor.

"stealing vehicle" scenario, fake iRobot positions were also given to try to get the vehicle to land in another location when iRobot was moving in the upper half part of the area. The monitor detected this anomaly through a violation of the invariant on the iRobot rate of change in position. In this case when there were the positions of iRobot changed two quickly, it kept flying without landing on either false or right platform. When iRobot was moving in the lower half part of cage it would try to land, which we considered a success. In the "false airport" scenario there was no iRobot, rather a false location was published. If the false location

was outside the region where the UAV had previously seen the iRobot, then the UAV refused to go to that location and filtered out these false messages. If the false location was in the correct range, the UAV attempted to land. However, the monitor could tell the difference of the height between the false and correct platforms, so the UAV with the monitor would not land on the false airport.

IV. Related Work

Our work was inspired in part by the evolution and maturity gained by techniques and tools available to infer likely program invariants. Our work builds specifically on Daikon [7], [8], one of the pioneer approaches with probably the most sophisticated toolset openly available [2]. Still, several other complementary efforts have emerged in the last few years, ranging from those attempting to integrate the inference and monitoring phases [13] to infer richer temporal properties such as event precedence [9], [16], and to use more static analysis such as symbolic execution to infer more general invariants that may hold for all scenarios or for certain program paths [6]. Our work is different from these efforts in its focus on messages, in the more domain-specific set of invariants we are trying to capture, and in the implementation of the toolset in the contexts of ROS so that invariants can not only be inferred but also included in a system to detect anomalies and take corrective measures.

In the context of robotic systems, monitoring for error detection is a well known area [15]. 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 grouped in analytical or data-driven, based on the system model used to detect anomalies. An analytical model derives properties from the physical world, while a data driven model uses input data to derive properties, usually of the statistical type based on the input. Just in the context of quad rotors similar to the ones we used in our study there have been several recent efforts that attempt to detect anomalies using data driven approaches [12] and more analytical ones combined with advanced machine learning approaches [11]. And even after an anomaly is detected existing efforts have been designed to perform diagnosis and remediation [10] based on models defined by domain experts. Our approach is complementary to these approaches, and unique in that it can generate more general invariants that were not considered by domain experts and not defined by simple statistics, and that may be relevant to many robotic systems as they are instantiated by a training set. Furthermore, the implementation within ROS makes it directly applicable to a large set of existing systems.

V. Conclusion and Future Work

We have introduced a general approach for automated invariant inference and monitoring, and implemented it in the context of ROS so that any system implemented with this operating system can leverage it with minimal effort. The approach was able to automatically infer rich invariants for a robotic system based on a training set, and it was able to detect the violation of those invariants and avoid failures under various scenarios of enough complexity to illustrate the potential of the approach.

Besides more extensive empirical assessment of the approach we see several technical avenues for future work. First, we would like to study how to increase the scalability of the approach. For invariant generation, we are investigating the application of filters based on the variance and pedigree of a variable as well as the automatic identification of redundant messages. Within invariants monitoring, we are investigating sampling schemes that can reduce the monitoring cost while minimizing information loss. As part of this effort we are studying how to incorporate training data from failing scenarios to further prioritize the invariants to monitor. Second, the approach generality and power could be increased by moving from invariants consisting of boolean expressions to probabilistic expressions, and by incorporating temporal operators, which may help to capture the uncertainty present in robotic systems. Last, the type of actions we support when an invariant is violated could be enriched to support, for example, message rectification so that minimally reformulated messages can be published but still remain within the system invariants.

References

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

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

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

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

[5] Ros. http://www.ros.org.

[6] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, pages 213–224, 1999.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. Mccamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, pages 35–45, 2006.

[9] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *FSE*, pages 339–349, 2008.

[10] F. W. Gerald Steinbauer, Martin Morth. Real-time diagnosis and repair of faults of robot control software. In *RoboCup*, pages 13–23, 2005.

[11] J. H. Gillula and C. J. Tomlin. Reducing conservativeness in safety guarantees by learning disturbances online: Iterated guaranteed safe online learning. In *RSS*, 2012.

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

[13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, 2002.

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

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

[16] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, pages 282–291, 2006.