A containerized ROS-compliant verification environment for robotic systems

Stefano Aldegheri, Nicola Bombieri, Samuele Germiniani, Federico Moschin, and Graziano Pravadelli
Department of Computer Science, University of Verona, Italy
Email: name.surname@univr.it

***Abstract*—This paper proposes an architecture and a related automatic flow to generate, orchestrate and deploy a ROS-compliant verification environment for robotic systems. The architecture enables assertion-based verification by exploiting monitors automatically synthesized from LTL assertions. The monitors are encapsulated in plug-and-play ROS nodes that do not require any modification to the system under verification (SUV). To guarantee both verification accuracy and real-time constraints of the system in a resource-constrained environment even after the monitor integration, we define a novel approach to move the monitor evaluation across the different layers of an edge-to-cloud computing platform. The verification environment is containerized for both cloud and edge computing using Docker to enable system portability and to handle, at run-time, the resources allocated for verification. The effectiveness and efficiency of the proposed architecture have been evaluated on a complex distributed system implementing a mobile robot path planner based on 3D simultaneous localization and mapping.**

***Index Terms*—ROS, verification, monitors, LTL, docker**

This research work has been partially supported by the project Dipartimenti di Eccellenza 2018-2022 funded by the Italian Ministry of Education, Universities and Research (MIUR) and INdAM – GNCS Project 2020

I. Introduction

One of the main impediments of efficiently developing large-scale and reliable robotic infrastructures with high-level autonomy is the lack of support for runtime system verification [1]. What is missing is a modular approach to verify, during the whole HW/SW design flow, the complex and heterogeneous software applications implementing the robot behaviors and tasks.

A reliable design practice requires the use of *monitors*, which are chunks of code that can check key properties of the system behavior in real-time, report violations, and possibly enforce fail-safe behaviors. Basic monitors to check resources and detect local faults for robotic applications are at the state-of-the art [2]. On the other hand, with an increased level of perception, interaction and control, current autonomous robotic systems need more advanced monitors to check their tasks. These include monitors for security and real-time safety constraints in addition to pattern matching over sensor readings to help perception. Various works have been proposed to generate these complex monitors automatically from their high-level specifications and integrate them into Robot Operating System (ROS)-based designs, for both single robots [3], [4] and robot swarms [5].

All these solutions are effective in generating verification environments where monitors are added to the System Under Verification (SUV) by assuming no limit concerning the computational resources available for its real-time execution. However, verification procedures introduce an overhead that, with the same amount of resources, could negatively impact the execution of the SUV tasks; i.e., SUV tasks can miss their real-time deadline since, now, part of the computational resources must be dedicated to verification. As a consequence, assertions monitoring real-time constraints can fail as well.

To address this problem, this paper proposes an assertion-based verification (ABV) architecture and a related automatic flow to generate, orchestrate and deploy monitors into a dynamic verification environment.

The rest of the paper is organized as follows. Section II details the problem statement. Section III describes the proposed verification architecture. Section IV is devoted to the monitor synthesis. Section V presents the orchestration approach. Section VI describes the experimental results. Finally, Section VII concludes the paper with final remarks.

II. Problem statement

We assume that the SUV is a real-time system composed of several software tasks running inside ROS-based nodes. ROS is the de facto standard middleware for developing robotic software applications. It implements messages passing among the system nodes by providing a *publish-subscribe* communication model. Every message sent by any node has a *topic*, which is a unique string known to all the communicating nodes. A node can create a topic to *publish* messages that will be received by all *subscribed* nodes.

The nodes execute on computational platforms distributed among different layers. The goal is to add a monitor-based architecture for semi-formal runtime verification of the SUV. The problem is how and where integrating monitors in the ROS-based system to achieve accurate verification without violating the real-time constraints of the SUV. More in detail, depending on how close each machine is to the source of the data, each ROS node is logically placed at a certain computational layer. The bottom layer is called “the edge”. Here we can find computing platforms such as microcontrollers and off-the-shelf devices. These devices guarantee low latency for monitor verification because data is elaborated on the spot. However, they provide low computation resources. Adding run-time verification at the edge may lead to violations of real-time constraints; indeed, the execution of monitors steals computational resources from software tasks, often preventing them from meeting their deadlines. The upper layer is called “the Cloud”. Here we have high-performance computing platforms, such as computer clusters and servers. Although the cost of

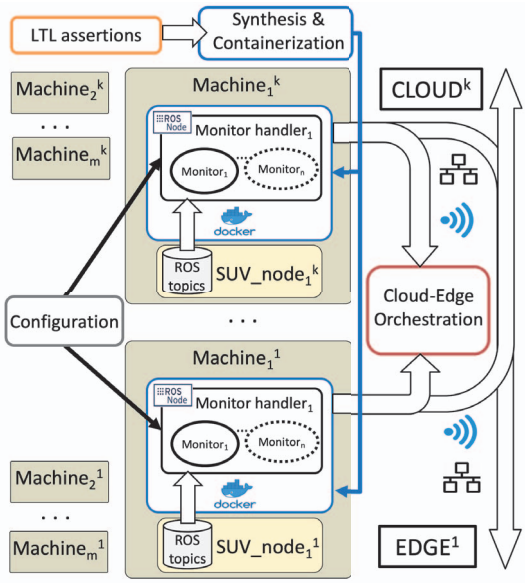

Fig. 1: Verification architecture.

monitor execution is negligible when carried out on these machines, the verification latency could become unpredictable and sometimes exceptionally high. This is due to the delay induced by moving data through the network from lower layers (where data is generated) to the cloud (where verification occurs), and vice-versa. As a result, notifications of failures may arrive too late.

III. Verification Architecture

To solve the issues described in the previous section, we propose the verification architecture depicted in Fig. 1. Our approach enables assertion-based verification of real-time robotic systems, by orchestrating the execution of runtime monitors among different layers, from the edge to the cloud. The verification workload is shared among the layers by considering constraints of latency, accuracy, and available computational resources. Monitors are automatically synthesized from linear temporal logic (LTL) assertions, usually following the template "always (antecedent $\rightarrow$ consequent)". Furthermore, they exploit the ROS publisher-subscriber paradigm to capture the events required by their evaluation function. Each *event* represents the value generated by the SUV for a variable at a specific time instant. For each variable used inside an assertion, a subscriber to the corresponding topic is created to receive the events in the synthesized monitor. Monitor verification is enabled in a plug-and-play fashion that does not require any modification of the SUV. Monitors execution is under the control of dedicated *monitor handlers*, one per each computing device in the system. Each handler is a ROS-compliant node containing both the orchestrator and the entire set of monitors. The verification environment is finally containerized into a Docker image to simplify the deployment of monitors across different hardware architectures and operating systems, and to handle the resources allocated for verification. We implemented a runtime system that dynamically migrates the execution of one or more monitors across the computing devices. It is worth noting that each machine contains a copy of the whole verification environment, yet only one instance of each monitor is executing in the system at any time. We report the details concerning monitor synthesis, containerization and orchestration in the next sections.

IV. Monitor Synthesis

This section describes how ROS-compliant monitors are automatically synthesized from LTL assertions. The input is a LTL assertion formalized by using the Property Specification Language (PSL). The output is a monitor composed of a C++ evaluation function to check the assertion dynamically, and a ROS-compliant infrastructure to capture the events to perform the evaluation. To better understand the process, let us refer to the running example of Fig. 2.

In the first step of the process, each proposition is substituted with a boolean variable acting as a placeholder. This is done because boolean values can be easily represented in a compressed format by using only one bit for each value. It becomes exceptionally convenient when implementing the monitor orchestration described in section V. In Fig. 2, the assertion $G(running \rightarrow speed < 100\ U\ stop)$ is substituted by $G(p_0 \rightarrow p_1\ U\ p_2)$, where p_0 is the placeholder for $running$, p_1 for $speed < 100$ and p_1 for $stop$.

Once the above substitution is completed, we generate a corresponding Büchi automaton. In Fig. 2, the generated automaton contains the states 0, 1 and 2. Finally, the automaton is visited to generate the monitor, which is mainly composed of a C++ evaluation function that checks the assertion at each simulation instant.

In the last step of the synthesis process, we create the ROS-compliant infrastructure to capture and handle the input for the evaluation function of the monitor. For each variable included in the evaluation function, we generate a callback to capture the related events from the corresponding topic. A callback function is attached to an independent thread executing each time a message is processed from the subscriber queue. An event is a couple $\langle new\_value, timestamp \rangle$ where new_value is the next value assumed by the variable and $timestamp$ is the instant at which the event was generated in the SUV. Each time a callback is called, the captured event is added to a buffer, i.e., it is not immediately processed by the evaluation function. As described later in Section V, the buffer allows the orchestrator to move the evaluation of the monitor across the edge-to-cloud layers, independently from the location of the machine where the events were observed. In addition, the buffer allows us to sort the events according to the order defined by their timestamps, thus minimizing evaluation errors due to synchronization issues and/or communication delays. The buffer is sorted each time its size reaches a certain threshold. A higher threshold ensures a better verification accuracy, as the events are more likely to be evaluated in the correct order. Once the buffer is sorted, the events are used to generate the values for the boolean placeholders. These values are stored in a compressed format in a new buffer. The compression process drops the timestamps (as the events are already ordered) and converts the boolean value to a single bit (1 for true, 0 for false). Each time the evaluation function is

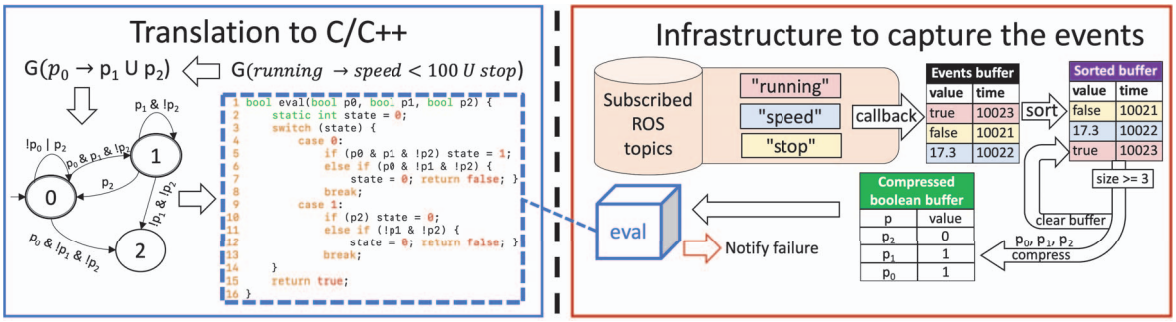

Fig. 2: Monitor synthesis and event subscription.

called, an event is consumed from the compressed buffer and used to advance the verification.

In Fig. 2, the ROS node is subscribed to three topics: running, speed, and stop. These topics correspond to the variables used in the assertion. In the example, three events are captured and added to the *event buffer* by the callback functions: $\langle 17.3, 10022\rangle$ for the numeric variable $speed$, and $\langle false, 10021\rangle$ $\langle true, 10023\rangle$ for the boolean variables $stop$ and $running$, respectively. Assuming that the sorting-threshold is set to 3, as soon as the third event arrives, the content of the *event buffer* is sorted according to the timestamps and moved to the *sorted buffer*. Then, each event is compressed and added to the *compressed boolean buffer*. Event $\langle 17.3, 10022\rangle$ is compressed to 1, as the corresponding proposition $p_1 : speed < 100$ is true for *speed* equal to 17.3, while $\langle false, 10021\rangle$ and $\langle true, 10023\rangle$ are directly translated to 0 and 1, respectively. Once the monitors are synthesized, they are *containerized* to enable portability of the verification environment across different HW/SW architectures. We extended the containerization procedure based on Docker for edge computing.

State-of-the-art solutions to containerized cloud-native applications associate each container to a private (isolated) subnet IP [6]. As a consequence, since ROS nodes cannot communicate if mapped to different subnet IPs, existing solutions allow communication of containerized ROS nodes only if all nodes are mapped to the same subnet IP. To solve this issue, which severely limits the applicability of such a container-based solution, the proposed architecture automatically maps the IP address of each container to the IP address of the host device (i..e, the device architecture in which the ROS node executes), while the port numbers are assigned randomly.

The proposed solution is modular, as it supports the easy generation and integration of containers for different HW/SW target architectures, from Cloud data centers to edge servers, off-the-shelf edge devices (e.g., NVIDIA Jetson), and embedded edge devices (e.g., robot native boards).

V. Monitor Orchestration

The purpose of the orchestration is to adjust, at run-time, the trade-off between verification responsiveness and resource consumption during the SUV execution.

Our orchestrating system consists in moving the monitor execution from a machine to another, possibly belonging to a higher computational layer, in the edge-to-cloud computing paradigm. This way we ensure that the receiving machine can

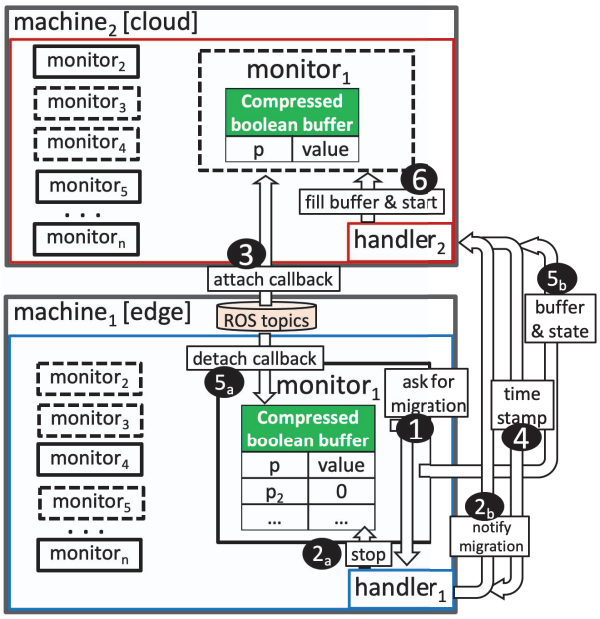

Fig. 3: Example of buffer migration.

provide the additional resources needed to handle the new monitor, at the expense of a reduction in responsiveness, as the evaluation of the monitor is located farther from the generation of the observed events.

Since each machine has a copy of all monitors, moving a monitor a_i from machine m_1 to machine m_2 can be achieved by (i) turning off a_1 on m_1, (ii) sending the state and event buffer of a_1 to $m2$ and (iii) activating a_1 on m_2. This procedure requires moving data through the network. However, the proposed migration approach is extremely efficient, as the data to be transferred are compressed, making the procedure exceptionally lightweight even for slow connections. Moreover, transferring the monitor's state is exceptionally efficient as well because it consists in just an integer value identifying the current state of the automaton implemented in the evaluation function.

We describe in detail the migration protocol between two machines hereafter. To simplify the exposition, let us consider the example shown in Fig. 3. The figure shows the movement of $monitor_1$ between $handler_1$ executing at level l_i of the edge-to-cloud hierarchy and $handler_2$ executing at level l_{i+1}. Before the migration is initiated, $handler_1$ is executing monitors 1 and 4, while $handler_2$ is executing monitors 2 and

5. Let us consider that, at a certain point, $monitor_1$ detects its event buffer is becoming full, as the machine cannot provide enough resources to consume the events. As a consequence, it asks $handler_1$ to be migrated (step 1). $handler_1$ removes $monitor_1$ from the scheduler (step 2_a), but it does not stop the process of adding events to its buffer. At the same time, it notifies the beginning of the migration to $handler_2$ (step 2_b). $handler_2$ attaches the correct callbacks to start adding events to the buffer of $monitor_2$ (step 3). However, $monitor_2$ is not yet put on the scheduler. Once $monitor_2$ receives enough events to perform the first sorting, $handler_2$ returns the timestamp of the oldest event in $monitor_1$ to $handler_1$ (step 4). This way, once $monitor_1$ receives the timestamp, it understands what events of the buffer must be sent, that is, all evaluated events with timestamp lower than the received timestamp.

When this happens, $handler_1$ detaches the callbacks from $monitor_1$ to stop the process of adding events to its buffer (step 5_a) and sends the correct events to $handler_2$ together with the state of $monitor_1$ (step 5_b). Now, $monitor_1$ is inactive on $handler_1$ and $handler_2$ finishes the migration by filling the buffer of $monitor_1$ with the received events and by putting it on the scheduler to start its evaluation (step 6).

VI. Experimental Results

A. Case Study

We applied the proposed methodology to a software application for an autonomous navigation mobile robot (Robotnik Kairos). The software implements an ORB-SLAM [7] application for localization and mapping through RGB cameras combined to an inference-based image recognition system [8] that controls a *move base* system based on a global and a local planner with obstacle avoidance.

Our goal was to verify the functional correctness of the application by considering the real-time constraints of the software system. In particular, we mapped the global planner to the cloud as a non-critical task. We considered the ORB-SLAM and local planner executing in real time at the edge on an NVIDIA Jetson TX2 and by setting a constraint of 8 FPS as the minimum supported rate for the RGB camera-input stream (i.e., 125 ms application makespan). Communication and synchronization between cloud and edge devices rely on the ROS software stack through Ethernet.

B. Results

In this subsection, we show that the proposed verification architecture can make the system endure computational loads that would be unfeasible with traditional assertion-based verification approaches. To accomplish that, we describe the results of simulating the case study with an increasing number of monitors, showing the improvements in applying our orchestration approach.

We considered the scenario where the robot has to reach a statically planned location $\langle x_2, y_2 \rangle$ from a starting point $\langle x_1, y_1 \rangle$. At first, the global planner finds a path to reach the arrival point.

After that, during the movement of the robot, the local planner reschedules the path trajectory (waypoints) according to the changes in the environment to avoid unwanted collisions with moving obstacles. We considered the 200 monitors initially allocated by the static orchestration to be executed on the Jetson TX2. During execution, we noticed that the monitor corresponding to the assertion $always((robot\_x_1 = x_1 \;\&\&\; robot\_y_1 = y_1 \;\&\&\; newGoal) \rightarrow (currentTime < timeOut \; U \; robot\_x_2 = x_2 \;\&\&\; robot\_y_2 = y_2))$ fails. The purpose of this monitor is to check if that robot arrives at $\langle x_2, y_2 \rangle$ before a certain timeout. The monitor failed because the overhead introduced by the verification environment caused an increase in the execution time. As a consequence, the controller of the robot could not support the minimum updating frequency of the motor velocities, making the robot move on the wrong trajectory. In particular, the controller needs at least an update of the motor velocities every 125 ms (8 Hz) to execute correctly. Without the verification environment, the system guarantees a makespan of 115 ms (8.7Hz). Such a value is increased to 140 ms (7.1 Hz) when including the verification overhead. By applying the buffer migration approach, the evaluation of 150 monitors was automatically moved from the Jetson to the Cloud during execution, freeing the edge device from some of the verification overhead. Because of this, the controller of the robot started working properly once again with a makespan of 125 ms (8 Hz), preventing the above assertion from failing.

This result shows that the approach is effective in freeing computational resources by moving the evaluation of monitors from congested computational devices.

VII. Conclusions

This paper proposed an architecture and a related automatic flow to generate, orchestrate and deploy a ROS-compliant verification environment for robotic systems. We provided several contributions regarding the synthesis, containerization and orchestrations of monitors. Execution of the architecture on a complex case study showed promising results toward a new comprehensive solution to apply assertion-based verification on real-time robotic systems.

References

[1] H. Abbas, I. Saha, Y. Shoukry, R. Ehlers, G. Fainekos, R. Gupta, R. Majumdar, and D. Ulus, "Special session: Embedded software for robotics: Challenges and future directions," 2018.

[2] F. Lorenz and H. Schlingloff, "Online-monitoring autonomous transport robots with an r-valued temporal logic," vol. 2018-August, 2018.

[3] P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the mop runtime verification framework," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2012.

[4] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications," *Lecture Notes in Computer Science*, vol. 10457, pp. 135–175, 2018.

[5] C. Hu, W. Dong, Y. Yang, H. Shi, and G. Zhou, "Runtime verification on hierarchical properties of ros-based robot swarms," *IEEE Transactions on Reliability*, vol. 69, no. 2, pp. 674–689, 2020.

[6] Docker, "Configure networking," 2020, https://docs.docker.com/network.

[7] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[8] NVIDIA, "Deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson," https://github.com/dusty-nv/jetson-inference.