AC-ROS: Assurance Case Driven Adaptation for the Robot Operating System

Betty H.C. Cheng*
chengb@msu.edu
Michigan State University
East Lansing, Michigan

Robert Jared Clark*
rjaredclark@gmail.com
Michigan State University
East Lansing, Michigan

Jonathon Emil Fleck*
fleckjo1@msu.edu
Michigan State University
East Lansing, Michigan

Michael Austin Langford*
langfo37@msu.edu
Michigan State University
East Lansing, Michigan

Philip K. McKinley*
mckinle3@msu.edu
Michigan State University
East Lansing, Michigan

ABSTRACT

Cyber-physical systems that implement self-adaptive behavior, such as autonomous robots, need to ensure that requirements remain satisfied across run-time adaptations. The Robot Operating System (ROS), a middleware infrastructure for robotic systems, is widely used in both research and industrial applications. However, ROS itself does not assure self-adaptive behavior. This paper introduces AC-ROS, which fills this gap by using assurance case models at run time to manage the self-adaptive operation of ROS-based systems. Assurance cases provide structured arguments that a system satisfies requirements and can be specified graphically with Goal Structuring Notation (GSN) models. AC-ROS uses GSN models to instantiate a ROS-based MAPE-K framework, which in turn uses these models at run time to assure system behavior adheres to requirements across adaptations. For this study, AC-ROS is implemented and tested on EvoRally, a 1:5-scale autonomous vehicle.

CCS CONCEPTS

• **Computing methodologies** → **Model development and analysis**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

KEYWORDS

self-adaptive systems, assurance case, cyber-physical systems, goal structuring notation, Robot Operating System, digital twin

ACM Reference Format:
Betty H.C. Cheng, Robert Jared Clark, Jonathon Emil Fleck, Michael Austin Langford, and Philip K. McKinley. 2020. AC-ROS: Assurance Case Driven Adaptation for the Robot Operating System. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3365438.3410952

*All authors contributed equally to this research.

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410952

1 INTRODUCTION

Increasingly, autonomic cyber-physical systems are applied in safety-critical application domains (e.g., automotive systems, aircraft, medical devices, power grids), where requirements violations can lead to property damage, physical injuries, and even loss of life. Assurance *certification* is one approach to ensuring correct operation of such systems, where *assurance cases* [24] are used to specify relationships between development artifacts (e.g., models and code) and the required evidence (e.g., test cases) needed to support a claim that the system operates as intended. However, many cyber-physical applications are built atop existing software frameworks that do not directly address software assurance. Therefore, a challenge is how to integrate assurance cases into such applications to certify they operate and adapt as expected. This paper demonstrates how assurance case driven self-adaptation can be realized in systems developed atop the Robot Operating System (ROS), a popular robotics middleware used in a wide variety of robotic systems, including autonomous vehicles.

Over the past decade, a number of research efforts have addressed assurance for self-adaptive systems [10, 47, 48], including the development of formal analysis techniques [8, 17, 51], formal modeling [23, 50], and proactively mitigating uncertainty [3, 37, 49]. The MAPE-K control loop [27] specifies a systematic approach to manage adaptations, but it does not explicitly address assurance [13]. For assurance certification purposes, the Goal Structuring Notation (GSN) has been developed to specify assurance cases graphically [44]. A GSN model defines specific claims for how requirements can be satisfied and what evidence is required to support those claims. Traditionally, the certification process and review of claims has largely been done manually [20, 26, 32, 43]. While work has been conducted to automate analysis of GSN models [15, 33], including security and safety properties analysis for self-adaptive systems [9, 25], GSNs have not been used to address assurance for self-adaptive ROS-based systems.

This paper introduces AC-ROS, a MAPE-K framework that uses GSN assurance case models to manage run-time adaptations for ROS-based systems. A key insight is that the modular ROS publish-subscribe infrastructure is ideal for (1) implementing the MAPE-K

control loop and event-driven adaptation and (2) supporting run-time monitoring of the environment and onboard system state. AC-ROS realizes the MAPE-K infrastructure in ROS, using GSN models at run time to guide adaptations that satisfy assurance requirements. Through automatic run-time analysis of GSN models, AC-ROS facilitates the development of assured, adaptive, and autonomous robotic systems.

This paper presents the main components of AC-ROS and describes a "proof-of-concept" implementation for EvoRally, a 1:5-scale off-road autonomous vehicle shown in Figure 1. The sensing, control, and actuation software of EvoRally is entirely ROS-based. In addition, a simulation of EvoRally provides a means to validate AC-ROS prior to deployment. Once validated, the ROS code used for simulation can be directly uploaded to EvoRally, thereby providing an assurance-driven *digital twinning* capability [30] for self-adaptive ROS-based systems. We demonstrate the operation of AC-ROS for an autonomous security patrol application where EvoRally is directed to survey the perimeter of an industrial facility by routinely visiting an established collection of waypoints.

Figure 1: EvoRally, a 1:5-scale vehicle constructed for this work, based on AutoRally, an open platform developed by researchers at Georgia Tech [19].

The remainder of this paper is organized as follows. Section 2 reviews background material for assurance cases, ROS, and self-adaptive systems. Section 3 overviews the EvoRally platform and the demonstration scenario. Section 4 describes the details of the AC-ROS framework. A proof-of-concept implementation of AC-ROS for EvoRally and illustrative adaptation scenarios are presented in Section 5. Section 6 discusses related work. Finally, Section 7 summarizes the results and suggests possible future directions.

2 BACKGROUND

This section provides background information on GSN, the operation of ROS, and the MAPE-K loop for self-adaptive software.

2.1 Goal Structuring Notation

Assurance cases provide a means to certify the ability of software to operate as intended [20]. In this approach, claims are made about how functional and non-functional requirements are met, and each claim requires supporting evidence for validation. One way to document an assurance case argument is through GSN modeling [44]. GSN allows an argument to be defined as a graphical model, with each claim represented as a *goal* and each piece of supporting evidence (e.g., test cases, documentation) represented as a *solution*. Additional elements, such as *assumptions*, *justifications*, *contexts*, and *strategies*, are provided within the notation to further expound upon an assurance argument.

A simple GSN example is shown in Figure 2 along with an abbreviated legend of selected elements and relationships defined by the GSN standard [44]. The figure depicts a single module (*M0*) of an assurance argument that claims a rover can successfully patrol its environment (*M0-G1*). Three sub-goals are included to support the top-level claim (*M0-G1.1*, *M0-G1.2*, and *M0-G1.3*, respectively). *M0-G1.1* claims that each waypoint is visited within an allotted timeframe, *M0-G1.2* claims that the rover avoids collisions, and *M0-G1.3* claims that the rover maintains a safe power level; *M0-G1.1* has an assumption, *M0-A1.1.1*, indicating that all the waypoints are reachable. Both *M0-G1.1* and *M0-G1.2* reference a separate module (*M1*), where additional supporting assurance arguments are defined. Strategies *M0-S1.1.1* and *M0-S1.3.1* provide scope and clarification on *how* supported claims should be resolved. Solutions *M0-S1.1.1.1.1* and *M0-S1.3.1.1.1* define supporting evidence.

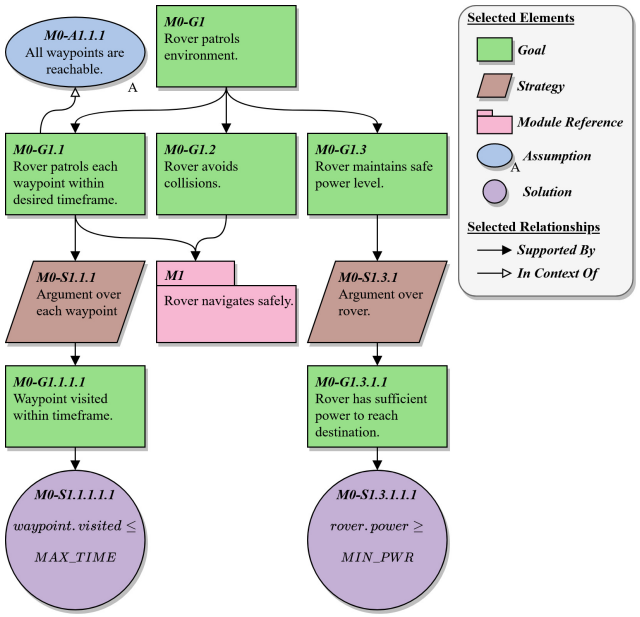


Figure 2: Example module of a GSN model. This module depicts an assurance argument for the claim that an autonomous rover can successfully patrol its environment.

As a graphical notation, GSN enables stakeholders with varying degrees of expertise to trace and audit the logic of an assurance case. In large-scale systems, however, the volume of documentation can become cumbersome and difficult to navigate manually [32], motivating recent work to explore automated GSN creation and refinement [15, 16], and automated GSN analysis [9, 25, 33]. Models have also been used at run time to manage self-adaptation [4, 5, 38], including assurance-focused approaches [12]. AC-ROS takes inspiration from these works to support a GSN-model driven approach to address assurance of a cyber-physical ROS-based system at run time.

2.2 Robot Operating System

The Robot Operating System (ROS) [40] is an open-source middleware platform created to support the development, testing and reuse of software for robots; its communication interface has become the *de facto* standard for robotic systems [36, 39]. A single ROS instance comprises a collection of ROS *nodes* (typically, Linux processes) that interact via a publish-subscribe paradigm over ROS *topics*. A topic is a unidirectional data stream (i.e., communication channel) to which multiple nodes can publish and subscribe. For example, a ROS node responsible for collecting raw data from a physical device, such as a wheel speed sensor, might preprocess data samples and then publish the modified stream on a relevant topic. Other interested nodes, including those responsible for different aspects of controlling the vehicle, can subscribe to this topic and use the received data for decision-making. In addition to topics, ROS also provides *services*, which implement request/reply interactions between pairs of nodes. ROS nodes are typically started via ROS *launch files*, which are configuration files that follow a specific XML format and include the name and parameters for each node to be started. ROS interactions can be visualized with *ROS graphs*. Figure 3 shows a simplified ROS graph for the sensing, control, and actuation of a rover.

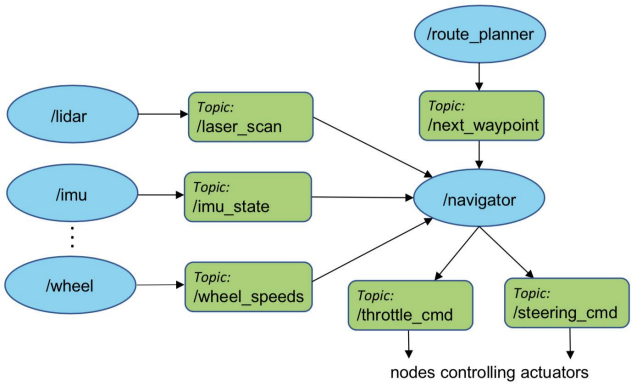


Figure 3: Simplified ROS graph for control of a rover (ROS nodes shown as blue ellipses and ROS topics as green boxes).

ROS is often coupled with the Gazebo physics-based simulator [29], which provides tested models of many commercially available hardware components. High-fidelity simulation enables the developer to explore and evaluate both high-level and low-level system behaviors at design time. A key feature of ROS is that the same ROS code developed for a simulated robot can be deployed and used to manage the corresponding physical robot. Over the past 10 years, ROS has gained a wide following among researchers and developers, and ROS is now used to control a variety of aerial, marine, and terrestrial robots [6, 45]. However, while a ROS developer may implement self-adaptive behavior for the robot, ROS itself does not assure that requirements are satisfied across adaptations.

2.3 MAPE-K Control Loop

The MAPE-K (Monitor-Analyze-Plan-Execute over shared Knowledge) control loop was introduced by Kephart and Chess [27] as a framework to manage adaptations for autonomic (self-*) systems. The four-step feedback loop begins by *Monitoring* the environment and system state. An *Analysis* of monitored data determines context for the current system state and selects an appropriate system adaptation. Next, *Planning* identifies an adaptation procedure based on the selected adaptation. Finally, *Execution* realizes the adaptation via system reconfiguration. Shared *Knowledge* includes information such as resource availability, performance constraints, and functional objectives. As described in Section 4, the MAPE-K loop is integral to AC-ROS by providing an automated means to use the requirements prescribed by the GSN model to manage the execution and adaptation of a ROS-based system.

3 DEMONSTRATION PLATFORM

Although AC-ROS is applicable to any ROS-based system, our initial target platform is EvoRally, whose hardware and software complexity enables testing of non-trivial scenarios involving dynamic adaptation. In describing the design and operation of AC-ROS in the following sections, we draw examples from the EvoRally implementation and an example surveillance application, depicted in Figure 4, where the vehicle is tasked with patrolling a physical space, for example, to help provide security at an industrial facility by periodically visiting waypoints along the perimeter of the facility.

The EvoRally vehicle is based on an open-source platform developed at Georgia Tech, called AutoRally [19], intended to facilitate research on autonomous vehicles without the cost and safety concerns associated with testing full-scale vehicles. The vehicle's chassis is borrowed from a gas-powered 1:5-scale remote-controlled truck. The engine is replaced with an electric motor, and many mechanical parts are replaced with physically stronger versions to accommodate the weight of computing equipment and sensors. A custom-built compute box houses a quad-core processor, 32GB RAM, 2TB SSD, and a GPU for real-time image processing. Sensors include a high-precision IMU, GPS, Hall-effect rotation sensor on each wheel, two front-facing machine vision cameras, and an optional Velodyne Puck lidar unit. The software infrastructure is entirely ROS-based, facilitating customization as well as integration of entirely new functionality, such as AC-ROS. The vehicle weighs 21 kilograms and has a top speed of 96 kilometers/hour.

Simulation is an important aspect of ROS-based development. In our work, we leverage and extend the concept of a *digital twin* [21, 30], where a physical system under development and/or post deployment is matched with a simulated equivalent to evaluate design alternatives and help reveal (and mitigate) potential problems. We have constructed a Gazebo-based digital twin of EvoRally by reusing and expanding simulation code developed for AutoRally. Here, we integrate AC-ROS into the control software of the EvoRally twin and use it to test and validate adaptive behavior in the context of the surveillance application. This code can then be transferred without modification to the physical EvoRally vehicle.

4 AC-ROS FRAMEWORK

This section provides detail about the core components and processes of the AC-ROS framework. The ROS design facilitated a modular and extensible realization of AC-ROS. Specifically, each

(a) Aerial photo of site for autonomous vehicle research.

(b) Simulated environment for patrolling rover application.

Figure 4: Scenario environment. The yellow boundary shows the perimeter of the space, purple circles show waypoints the rover must periodically visit, and the blue square shows the base station where the rover can be stored and obtain service.

MAPE-K processing element can be implemented as one or more ROS nodes, with communication implemented as ROS topics. Moreover, our approach provides a means to systematically integrate assurance information from GSN models with ROS- and platform-specific information to initialize MAPE-K elements and guide run-time monitoring and adaptations. As such, AC-ROS enables ROS-based platforms to conform to GSN models at run time, thereby assuring that the system continues to satisfy requirements while adapting as necessary. The remainder of this section describes how GSN models are created and analyzed at design time, as well as how they are used at run time to manage the adaptations. For clarity, we use the following font conventions: GSN modeling elements are denoted in italics with *MX-YY* labels, ROS code elements are denoted in courier, and AC-ROS elements are capitalized and italicized.

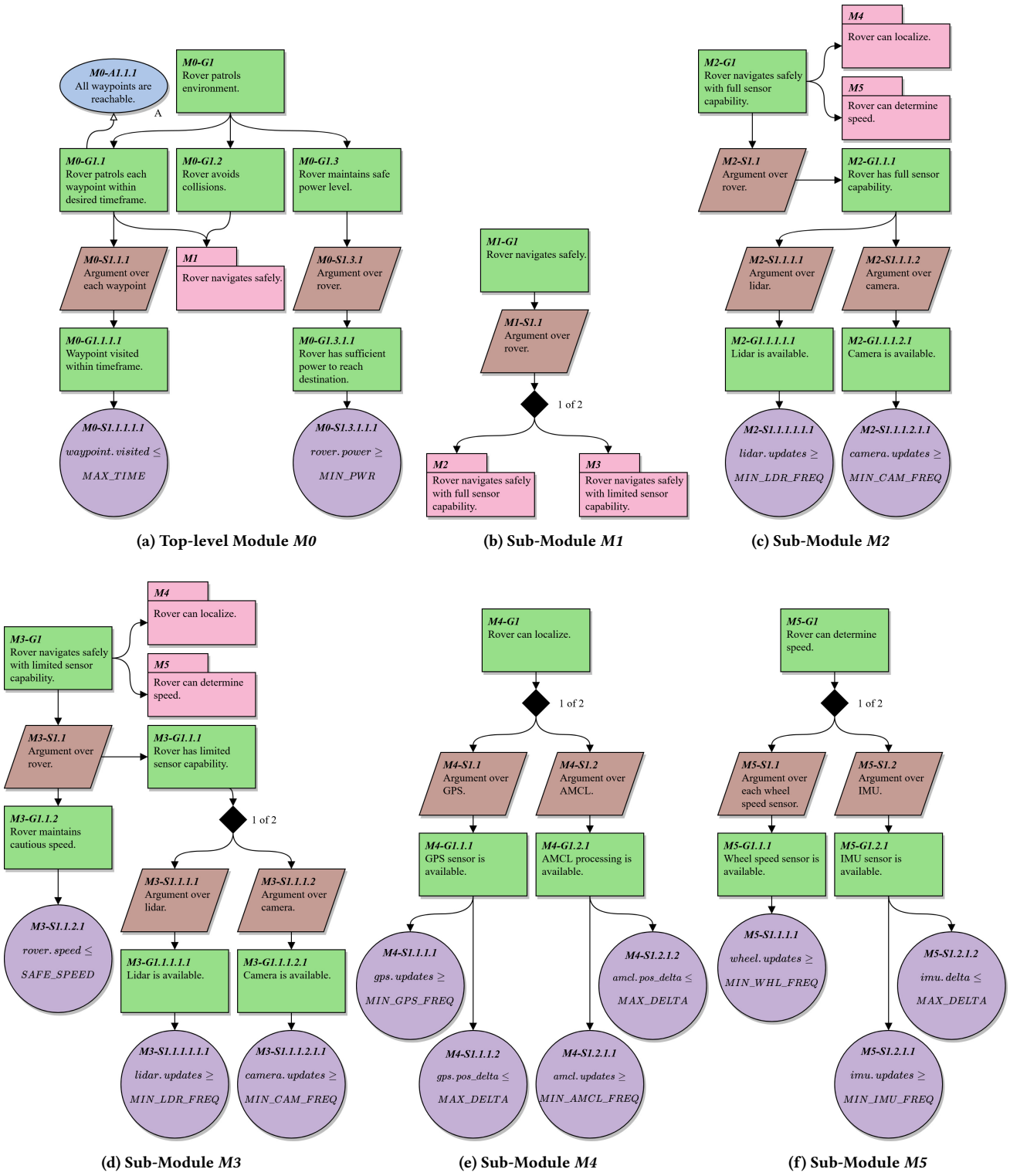
4.1 GSN Modeling in AC-ROS

Since our approach requires an assurance case to be developed for the autonomous platform in the form of a GSN model, we have developed two supporting tools for this work: GSN-Draw and GSN-Interpret. GSN-Draw is a graphical user interface to enable developers to construct an assurance case with standard GSN (Version 2) conventions [44]. GSN-Interpret is a Python code library that enables parsing and run-time evaluation of GSN models produced by GSN-Draw.

GSN-Draw has been implemented as a web application to enable the construction of a GSN model. GSN elements can be selected and instantiated into a "drawing" pane. Instantiated elements can then be linked together by any supported relationship type to form a connected graph. Constructed GSN models can be exported into a parsable XML format.

The GSN standard allows for a single GSN model to be composed of multiple interconnected modules. These modules are developed individually within a given project using the GSN-Draw tool. Figure 5 depicts several aggregate GSN modules that describe the assurance case for the EvoRally platform. For example, Figure 5(a) shows the top-level module (*M0*) of a GSN model created for an autonomous rover, whose mission is to visit a set of waypoints. This module makes the most general claim (*M0-G1*) about a rover's ability to complete its mission. A supporting argument is expressed by all elements connecting to *M0-G1*. Additional (sub-)modules, *M1* through *M5* (depicted in Figures 5(b) through 5(f), respectively) are developed in a similar manner but constructed separately to improve readability and enable the reuse of common sub-arguments. (Pink-tabbed boxes refer to sub-modules for a given project.) The assurance argument laid out by a GSN model must be supported by evidence. Solution elements (such as *M0-S1.1.1.1.1* in Figure 5(a)) state what specific evidence is required to satisfy parent claims. For this work, solution elements are expressed by *utility functions* [14, 28, 37, 46]. Utility functions are derived from system and/or environmental properties that can be monitored at run time to determine the satisfaction of system requirements [41]. The utility function for *M0-S1.1.1.1.1* checks the last time each waypoint was visited by the rover, and it is satisfied when each waypoint has been visited within the maximum allotted time. In a similar manner, the entire GSN model can be assessed by evaluating all utility functions against the current state of the rover. Once all utility functions have been evaluated, graph tracing can determine if the root-level claim is satisfied.

Other assurance case editing tools provide varying degrees of support for the GSN standard [34]. A primary motivation for developing GSN-Draw is to facilitate the use of utility functions as evidence for GSN solution elements, which is not supported by existing tools. Additionally, for adaptive systems, it is expected that assurance cases will include *optional* branches and alternative strategies for assuring different system adaptations. GSN-Draw enables a GSN model to be created fully compliant with the GSN (Version 2) standard, including the extensions to support argument patterns (i.e., optionality and modules).



(a) Top-level Module *M0* **(b) Sub-Module *M1*** **(c) Sub-Module *M2***

(d) Sub-Module *M3* **(e) Sub-Module *M4*** **(f) Sub-Module *M5***

Figure 5: A standard GSN model of an assurance case for a patrolling rover. (a) Module *M0* argues that a rover can successfully patrol its environment. (b) *M1* argues that a rover can navigate its environment safely. (c) *M2* argues that a rover can safely navigate with full sensor capabilities. (d) *M3* argues that a rover can safely navigate with only limited sensor capabilities. (e) *M4* argues that a rover can localize its position. (f) *M5* argues that a rover can determine its speed.

4.2 Knowledge Base

The AC-ROS *Knowledge Base* is an aggregate collection of static and dynamic data stores relevant to system-specific and GSN-modeled assurance information. For brevity, we use the prefix 'KB:' to refer to individual *Knowledge Base* elements; we use *Knowledge Base* when referring to the entire AC-ROS knowledge base.

KB1. GSN Model. The GSN model is exported from GSN-Draw to this knowledge base element, which will be used at run time to manage adaptations.

KB2. Mapping to ROS Topics. This data store associates every utility function parameter referenced by the GSN model to a corresponding ROS topic for the target platform. For example, the utility function in *M0-S1.1.1.1.1* in Figure 5(a) references a *rover.power* parameter, which is linked to the `charge_percent` ROS topic via this mapping. This mapping is not generated automatically, since it is platform-specific and requires knowledge of the available ROS topics.

KB3. ROS Launch Files. This data store contains information needed to configure the run-time monitoring processes for AC-ROS. The ROS launch files are automatically generated and contain names of *Monitor* nodes and relevant ROS topics based on the mapping from the utility function parameters.

KB4. Adaptation Tactics. This data store is a collection of adaptation strategies [14], each of which is described by its *context* (i.e., conditions for when to apply a given adaptation), sequence of *actions*, and consequences of adaptation (i.e., effect(s) of adaptation). Each adaptation tactic is a high-level sequence of commands to enable changes in behavior 'modes' (e.g., a mode change could switch sensors from cameras to lidar for obstacle detection). These tactics are described by configuration files that are loaded at the beginning of run time.

KB5. System State. This store contains run-time monitored system information supplied by the *Monitor* nodes. Entries for system components include raw data (e.g., wheel speeds), meta information (e.g., frequency and distributions of data stream samples), and various configuration parameters (e.g., settings for obstacle detection mode).

These *Knowledge Base* elements are discussed in the following descriptions of AC-ROS design-time processing and run-time operation.

4.3 Design-time Activities

The data flow diagram (DFD) in Figure 6 overviews the design-time steps for modeling an assurance case and using it to configure the run-time ROS environment. Descriptions of these four Steps D1-D4 follow.

Step D1. Construct GSN Model. GSN-Draw enables a requirements engineer to interactively develop a graphical model for an assurance case in the form of one or more GSN modules.

Step D2. Export GSN Model. GSN-Draw exports a collection of XML files to **KB1**, each of which correspond to a respective GSN module. Exported files preserve the structure and contents of the

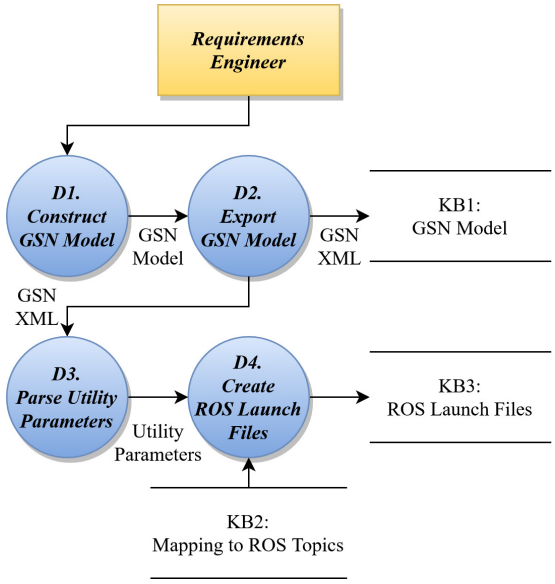


Figure 6: Data flow diagram for creating a GSN model and initializing the ROS environment for AC-ROS at design time. Circles, boxes, arrows, and parallel lines, respectively represent processes, external entities, data flow, and data stores.

GSN model, including element properties, relationship dependencies, and associated utility functions.

Step D3. Parse Utility Parameters. GSN-Interpret reads XML files exported by GSN-Draw, isolates the utility functions associated with each solution of the GSN model (e.g., rover remaining power is greater than minimum power level allowed), and then extracts referenced functional parameters (e.g., rover power level needs to be monitored).

Step D4. Create ROS Launch Files. While the run-time monitoring activities conducted by AC-ROS are defined by the utility function parameters specified in the GSN model, they are realized by a collection of ROS nodes that subscribe to relevant ROS topics. This step creates ROS launch files that define the ROS nodes to be instantiated, to which topic each should subscribe, and the corresponding utility function in the GSN model. The created launch files are stored in **KB3** and later used to configure run-time monitors. The mapping between utility parameters and ROS topics are provided by developer via **KB2** (e.g., utility function parameter for rover power is mapped to ROS topic for the onboard battery power level sensor). This use of indirection and abstraction enables the requirements engineer to design a GSN model without needing to know implementation details of the target ROS platform, thereby facilitating reuse of the GSN model for alternative ROS platforms.

4.4 Assuring System Adaptations at Run time

Figure 7 depicts the run-time elements, Steps R1-R4, that comprise the AC-ROS MAPE-K loop. Collectively, these elements manage system adaptations that satisfy an assurance case for a ROS-based autonomous platform. Descriptions for each diagram element follow.

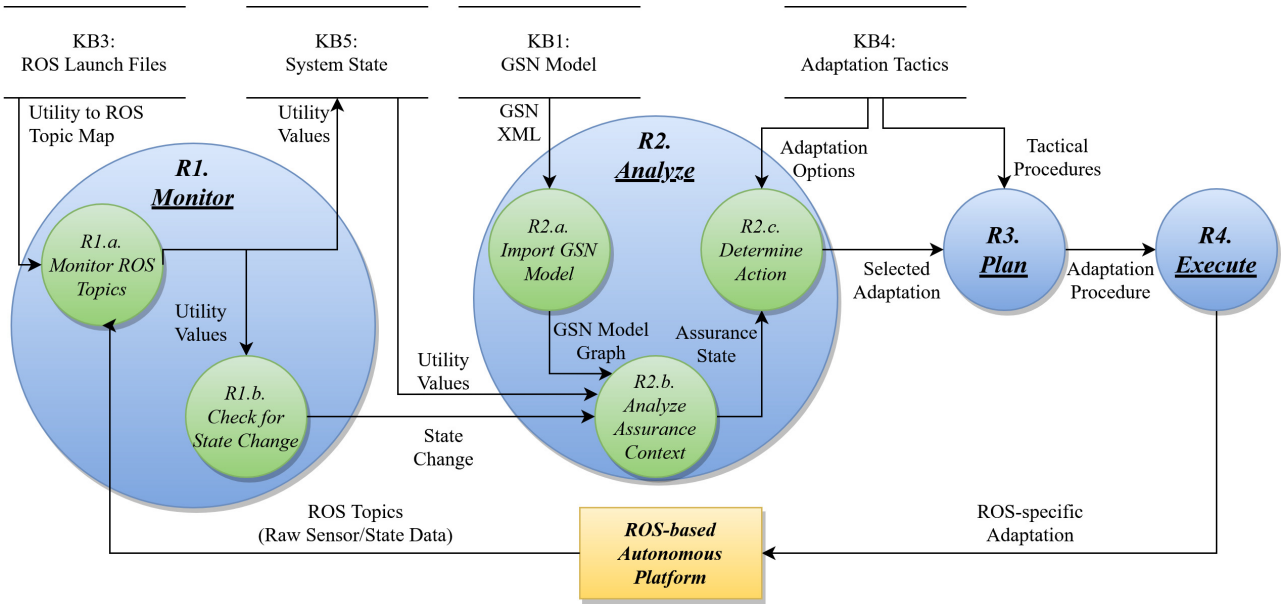


Figure 7: Data flow diagram for run-time processes involved with managing adaptations of a ROS-based platform.

Step R1. Monitor. The *Monitor Step* of the AC-ROS framework comprises one or more *Monitor* nodes that observe the managed autonomous platform and update **KB5** accordingly. They are also capable of performing minor preprocessing to gather aspects of individual components (e.g., frequency, inferred speed from GPS measurements, etc.). A single *Monitor* node is responsible for a given subsystem (e.g., lidar, camera) which, for this study, is represented by a single ROS topic. When instantiated, *Monitor* processes refer to command-line parameters in respective launch files for information on which target subsystem to observe, the relevant ROS topic, what aspects to record, and the associated utility function referenced by the GSN model (*Step R1.a.*). As each *Monitor* process observes its subsystem and updates **KB5**, it also checks for any violations of its associated utility function (*Step R1.b.*). A utility function violation serves as an *adaptation trigger*, which then activates the *Analyze Step* accordingly.

Monitor processes are instantiated to account for every corresponding utility function parameter referenced by the GSN model, including all graph branches in the case of optionality. During run time, it is assumed all optional branches of the GSN model graph are available to be satisfied and are thus monitored. However, if needed AC-ROS supports adaptive monitoring by allowing nodes to subscribe to, and unsubscribe from, the relevant ROS topics during run time.

Step R2. Analyze. The *Analyze Step* determines whether adaptation is needed to ensure that the system delivers acceptable behavior as specified by the assurance case. Upon startup of the *Analyze Step*, the GSN model is loaded from XML files and stored in an internal graph representation (*Step R2.a.*). The source XML specifies a root goal element for each GSN module and parent-child relationships for each GSN element. Subgraphs are constructed for each module to preserve the specified relationships from root to leaf nodes. Links are then established to the module subgraphs in order to produce an internal graph of the entire GSN model.

The *Analyze Step* is activated periodically, as well as when one of the *Monitor* processes reports a violation of a utility function. The *Analyze Step* retrieves current state information from **KB5** and GSN-Interpret to determine if the managed system complies with the imported GSN model (*Step R2.b.*). GSN-Interpret evaluates the GSN graph via a depth-first traversal to determine the satisfaction of leaf-level solution elements based on the evaluation of associated utility functions. Example utility functions can be found in the leaf nodes of GSN modules in Figure 5. Based on this evaluation of the GSN model, the *Analyze Step* determines whether any adaptations are necessary. When the GSN model is not satisfied, the existing adaptation tactics are reviewed based on the descriptions of their contexts and consequences of adaptation impact.

For this work, it is assumed that corresponding adaptation tactics have been predefined to cover each combination of utility function violations. Ongoing research includes the use of reinforcement learning to determine which adaptation tactics are best suited for the given context. Once an appropriate adaptation tactic has been identified (*Step R2.c.*), the *Analyze* process passes a reference to the selected adaptation to the *Plan Step*.

Step R3. Plan. The *Plan Step* receives the reference to the selected adaptation tactic and retrieves the corresponding sequence of actions from **KB4**. The sequence of actions for a given adaptation tactic ensures that the managed system is in a quiescent state before the system adapts [31]. For example, a tactic for changing obstacle detection modes might require that the robot slow down first to reduce the possibility of collision.

Step R4. Execute. The *Execute Step* receives adaptation actions from the *Plan Step* and translates them into platform-specific commands, which it publishes on ROS topics subscribed to by nodes that carry out the adaptation.

4.5 Realization in ROS

As a proof-of-concept, we have implemented and tested the AC-ROS framework in the EvoRally digital twin. As with most ROS systems, EvoRally's ROS infrastructure runs atop Linux. ROS execution begins with `roscore`, a collection of nodes that start/manage other ROS nodes and implement publish-subscribe communication on ROS topics. Other ROS nodes are launched as individual Linux processes according to their respective launch files. In total, the EvoRally ROS infrastructure comprises 35 ROS nodes and 278 ROS topics, of which 12 nodes and 7 topics comprise the current AC-ROS implementation.

Although the complete ROS graph for EvoRally is too large for inclusion in the paper, the ROS implementation of the MAPE-K loop is illustrated in Figure 8. The `monitor_xxxx` nodes are started from launch files created by *Step D4* of Figure 6. The other MAPE processes are implemented as individual ROS nodes, namely, `analyze`, `plan`, and `execute`. In addition, a `knowledge` node is created in order to manage the entirety of the AC-ROS *Knowledge Base* and thus support updating system-state information and retrieval of the GSN model with utility functions and adaptation tactics. The `knowledge` node subscribes to three `update_entry` topics, on which `monitor` nodes publish system state updates; each of the `update_entry` topics handles a particular data type (int, float, and string). The `knowledge` node also provides a ROS service, `get_entry`, which other nodes can use to obtain data in a request-reply manner. For example, `analyze` and `plan` nodes can request the retrieval of information for a given adaptation tactic from the `knowledge` node.

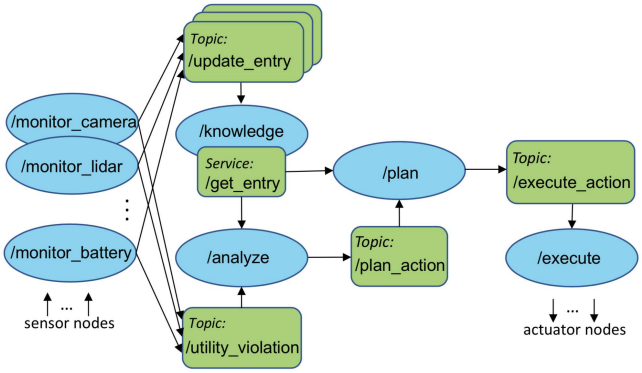


Figure 8: Simplified ROS graph for MAPE-K loop implemented on EvoRally. ROS nodes shown as blue ellipses. Data flows between nodes over ROS topics, shown as green boxes.

As described above, each `monitor` node implements one or more of the utility functions in the GSN model. If the function evaluates to false, then the corresponding `monitor` node publishes this information on the `utility_violation` topic, triggering the `analyze` node to re-evaluate the GSN model to determine if the top-level goal is still satisfied, or whether an adaptation is needed. It does so by invoking GSN-Interpret methods to evaluate the rover's compliance with the GSN model (*Step R2* above). In the current implementation, the `analyze` node is the only element of the MAPE-K loop that has direct access to the GSN model. Any selected adaptation is communicated to the `plan` node via the `plan_action` topic. The `plan` node acquires the relevant adaptation tactic from the `knowledge` node via the `get_entry` service and forwards it to the `execute` node via the `execute_action` topic. The `execute` node translates the adaptation actions into platform-specific commands for underlying actuator nodes.

5 DEMONSTRATION

In this section we demonstrate how AC-ROS governs dynamic adaptation in the application described in Section 3, where EvoRally is responsible for patrolling the perimeter of an area shown in Figure 4. We begin by providing necessary background on the operation of the vehicle.

5.1 EvoRally Operation

A particularly important part of the EvoRally ROS infrastructure is the *navigation stack*, a collection of nodes that cooperate to guide the vehicle from its current *pose* (location and orientation) to a goal pose. The navigation stack maintains both a global and local cost map of the environment, allowing it to plan paths to goal locations while avoiding obstacles. A cost map can be thought of as an occupancy grid that indicates the degree to which a cell is empty or occupied. The `move_base` node uses these maps in deciding how to move toward the goal pose, then publishes linear and angular velocity (a.k.a. *twist*) commands, which are converted by a `base_controller` node into low-level steering servo and electric motor commands. The current goal pose is published by a `waypoint_publisher` node. As each waypoint is reached, the `move_base` node publishes a success message, which `waypoint_publisher` uses as a trigger to publish the next waypoint.

To determine its current pose, the vehicle relies on a *localization* algorithm. For this study, EvoRally uses Advanced Monte Carlo Localization (AMCL) [18, 35], where a particle filter is used to estimate the robot's position and orientation based on a known map of the surroundings. AMCL is sufficient for localization in our surveillance/patrolling application, since a map of the area can be constructed prior to deployment. Other applications, such as search and rescue in unfamiliar terrain might use GPS for localization. The ROS AMCL package is designed to process only laser scans from lidar units. Since EvoRally is also equipped with stereo cameras and an accompanying ROS stereo vision package, `stereo_image_proc`, we developed code that converts point clouds produced by `stereo_image_proc` into laser scans that can be used by the AMCL package. As a result, the AMCL package on EvoRally can work with laser scans from either the lidar or the cameras.

Figure 9 shows three views of the EvoRally digital twin navigating the simulated test environment using lidar and cameras. The camera-generated data becomes considerably noisier and less accurate than the lidar data as distance from the vehicle increases,

as shown in Figure 9(c). Also, the cameras are restricted to a relatively narrow field of view, while the lidar has a 180° field of view. This difference is illustrated by the obstacle directly to the left of EvoRally in Figure 9(a): as shown in Figure 9(c), the lidar detects the obstacle but the stereo cameras do not. Due to these differences, by default AMCL uses the lidar laser scan instead of the stereo cameras. The GSN model requires the vehicle to proceed more cautiously (e.g., lower maximum speed) when operating in camera-only mode (see *M3-i*n Figure 5(d)).

EvoRally is powered by two Lithium-Polymer battery banks. The compute box bank is a single 6s 11000 mAh LiPo battery, while the chassis bank comprises two 4s 6500 mAh LiPo batteries connected in series. Here we are concerned primarily with the chassis batteries, which have a typical run time of one hour, depending on usage. The level of charge in the chassis batteries is available from the electric motor's speed controller. However, since the speed controller is a physical device that does not exist in simulation, for the digital twin we developed a ROS node, `battery_sim`, to simulate drain on the battery as a function of the vehicle's speed. The `battery_sim` node publishes the remaining charge as a percentage on the `charge_percent` topic. The GSN model in Figure 5(a) includes a solution element that references the *rover.power* utility parameter, which is mapped to the `charge_percent` topic in *Step D4* of Figure 6 to create a launch file for a corresponding `monitor_battery` node. The `monitor_battery` node subscribes to the `charge_percent` topic and forwards updated values to the *Knowledge Base* via the `update_entry` topic.

5.2 Run-time Adaptations

Now let us consider the patrolling application. Assume the rover begins its mission, departing the base station with the first waypoint, W1, as the destination for the `move_base` node. The rate at which the simulated chassis battery level drains increases as the rover's speed increases, with the remaining charge monitored by the `monitor_battery` node. The `move_base` node continues to publish twist commands to direct the rover toward W1 based on localization estimates from the `AMCL` node, until W1 is reached and the `waypoint_publisher` publishes the next waypoint, W2. This procedure continues for the remaining waypoints, after which `waypoint_publisher` starts the next lap by publishing W1.

Eventually, the battery level drops below the minimum power level defined in the solution node *M0-S1.3.1.1.1*, causing the utility function implemented by `monitor_battery` to evaluate to false. The `monitor_battery` node reports this violation by publishing a message to the `utility_violation` topic. The `analyze` node receives this message and submits a `get_entry` service call to obtain updated values for all monitored items in GSN utility functions.

Once this information is received, the `analyze` node uses GSN-Interpret to validate the entire GSN model with the updated parameters received from the service call. The `analyze` node parses the GSN-Interpret results and determines that the goal *M0-G1.3* is not satisfied because the corresponding solution *M0-G1.3.1.1* is violated (Figure 5(a)). By searching the allowed adaptation tactics, the `analyze` node determines that the only relevant adaptation is to adjust the patrol mode so that EvoRally returns to the base station (*Step R2.c.* in Figure 7). It sends the corresponding tactic context via the `plan_action` topic to the `plan` node, which expands the tactic into a sequence of actions that will produce the desired adaptation (*Step R3* in Figure 7), and publishes this sequence on the `execute_action` topic. Upon receiving this information, the `execute` node translates the steps into ROS messages and publishes them on the corresponding topics (*Step R4* in Figure 7). In turn, the `waypoint_publisher` publishes the location of the base station as the desired goal, which causes EvoRally to change course and proceed toward the base station.

Next let us assume that while EvoRally is enroute to the base station, the `monitor_lidar` node detects that the rate of messages arriving on the `lidar_front` topic has dropped below the prescribed threshold, *MIN_LDR_FREQ*; the corresponding variable is referenced in the GSN's utility functions as *lidar.freq* (Figures 5(c) and 5(d)). As with the low battery adaptation, the `analysis` node parses the results from GSN-Interpret and determines that the violation can be resolved in two different ways: either satisfy the lidar solution by increasing *lidar.freq* or switch to using the camera-generated laser scans for localization and obstacle detection. In searching the allowed tactics, only the latter context is found. The `plan` and `execute` nodes expand and translate this option into a set of ROS messages published on relevant ROS topics. The vehicle then proceeds to the base station, using only its cameras for navigation, for battery replacement/recharging and possible repair or replacement of the lidar unit. This option is represented in Sub-Module M3 of the GSN model (Figure 5(d)). Experimentally, the run-time performance of executing the `analyze`, `plan`, and `execute` nodes for analyzing the EvoRally GSN model and using it to codify and carry out the adaptation cumulatively requires less than 3 milliseconds per adaptation.

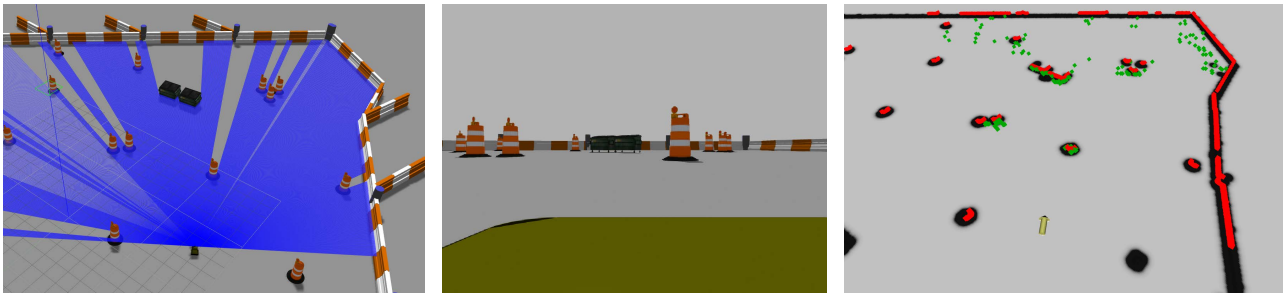
6 RELATED WORK

While other work has explored model engineering with ROS [2, 7, 22] and self-adaptation with ROS [1, 11], our work specifically brings together self-adaptation with MAPE-K feedback to support assurance through GSN assurance cases for ROS-based systems.

Calinescu et al. introduced ENTRUST [9], a method for assurance-driven adaptation. Like AC-ROS, ENTRUST implements a MAPE-K control loop for self-adaptive systems, guided by GSN models. However, during the analysis step of the MAPE-K loop, ENTRUST uses a probabilistic verification engine to assess stochastic finite state transition models (i.e., Markov chains) of the controlled system. In contrast, AC-ROS has been designed to take advantage of ROS, and instead of relying on a model of the autonomous platform for assurance assessments, AC-ROS directly monitors real system data via ROS topics and evaluates assurance arguments based on utility functions.

The RoCS framework [42], created by Ramos et al., implements a MAPE-K control loop to manage adaptations for robotic systems, with potential for integration into ROS. However, a key difference between RoCS and AC-ROS is that AC-ROS uses GSN models to guide and assure adaptations at run time. While RoCS and AC-ROS both realize the MAPE-K control loop, AC-ROS ensures that safety requirements are upheld with each executed system adaptation.

Alternative methods to automated evaluation of assurance cases have been explored. Lin et al. [33] introduced a method that uses



(a) (b) (c)

Figure 9: Demonstration scenario simulated in Gazebo: (a) overhead view showing lidar (blue) scan; (b) view from the perspective of the rover; (c) visualization of lidar (red) and camera-generated (green) laser scans, rover position is shown as a yellow arrow, and true obstacle locations are shown in black.

Dempster-Shafer theory to calculate confidence for an assurance case. AC-ROS acquires evidence directly via the evaluation of utility functions, and thus confidence is not explicitly considered when evaluating evidence for an assurance case. Traditionally, assurance cases have focused on certifying functional requirements for the purpose of safety, although recent work has addressed assurance for security requirements [25]. Thus far, AC-ROS has been implemented to manage only functional assurance cases. However, for future work, we plan to extend the framework to include consideration for security assurance cases.

7 CONCLUSIONS AND FUTURE DIRECTIONS

This paper introduces the AC-ROS framework to support an assurance case driven approach to developing and managing ROS-based adaptive systems. We leverage the ROS publish-subscribe communication paradigm to implement the AC-ROS MAPE-K feedback loop to support (GSN) model at run-time driven adaptation. We also developed two supporting tools. GSN-Draw can be used by developers to create graphical GSN models for assurance cases that capture system requirements and supporting assurance information. GSN-Interpret supports the run-time assessment of the GSN models for compliance to requirements according to utility functions that capture relevant system and environmental properties.

A proof-of-concept realization of the AC-ROS framework has been developed and applied to manage adaptations for EvoRally, a 1:5 scale autonomous rover. Scenarios illustrate how AC-ROS can be used to handle changing (adverse) system and environmental conditions that warrant adaptation.

We are pursuing several lines of research related to the AC-ROS framework. For example, we are exploring how machine learning techniques, especially reinforcement learning, can be used to support the AC-ROS *Analyze Step* to further enhance the resiliency of a system. Also, we are exploring how search-based techniques (e.g., ML and evolutionary-based) can be used to "harden" the system requirements and other GSN modeled elements that drive the AC-ROS process. Finally, our longer-term objective is to build an assurance-centric digital twin framework for cyber-physical systems, including support for a full-scale autonomous vehicle running ROS.

ACKNOWLEDGMENTS

We greatly appreciate the contributions of Glen A. Simon to previous work on this project. This work has been supported in part by grants from NSF (CNS-1305358 and DBI-0939454), Ford Motor Company, and General Motors Research; and the research has also been sponsored by Air Force Research Laboratory (AFRL) under agreements FA8750-16-2-0284 and FA8750-19-2-0002. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL), the U.S. Government, National Science Foundation, Ford, GM, or other research sponsors.

REFERENCES

[1] Jonathan M. Aitken, Sandor M. Veres, and Mark Judge. 2014. Adaptation of System Configuration under the Robot Operating System. *IFAC Proceedings Volumes* 47, 3 (2014), 4484–4492. https://doi.org/10.3182/20140824-6-ZA-1003.02531 19th IFAC World Congress.

[2] Gianluca Bardaro, Andrea Semprebon, and Matteo Matteucci. 2018. A Use Case in Model-Based Robot Development Using AADL and ROS. In *Proceedings of the 1st International Workshop on Robotics Software Engineering (RoSE '18)*. ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/3196558.3196560

[3] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. 2010. Fuzzy Goals for Requirements-Driven Adaptation. In *Proceedings of the 18th IEEE International Requirements Engineering Conference*. IEEE Computer Society, Washington, DC, USA, 125–134.

[4] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@run.time: A Guided Tour of the State of the Art and Research Challenges. *Software and Systems Modeling* 18, 5 (01 2019), 3049–3082. https://doi.org/10.1007/s10270-018-00712-x

[5] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@ Run.Time. *Computer* 42, 10 (Oct 2009), 22–27.

[6] Aaron Blasdel and et al. 2020. ROS Robots. Showcase of robots using ROS, available at http://robots.ros.org.

[7] Etienne Borde, Grégory Haïk, and Laurent Pautet. 2009. Mode-Based Reconfiguration of Critical Software Component Architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*. European Design and Automation Association, Leuven, BEL, 1160–1165.

[8] Radu Calinescu, Carlo Ghezzi, Marta Z. Kwiatkowska, and Raffaela Mirandola. 2012. Self-Adaptive Software Needs Quantitative Verification at Runtime. *Commun. ACM* 55, 9 (2012), 69–77.

[9] Radu Calinescu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli, and Tim Kelly. 2018. Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1039–1069.

[10] Javier Cámara, Rogério de Lemos, Carlo Ghezzi, and Antónia Lopes (Eds.). 2013. *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*. Vol. 7740. Springer, Cham, DEU.

[11] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carreraa, Narcís Palomeras, Natàlia Hurtós, and Marc Carrerasa. 2015. ROSPlan: Planning in the Robot Operating System. In *Proceedings of the Twenty-Fifth International Conference on International Conference on Automated Planning and Scheduling (ICAPS '15)*. AAAI Press, Palo Alto, CA, USA, 333–341.

[12] Betty H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. 2011. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time - Foundations, Applications, and Roadmaps [Dagstuhl Seminar 11481, November 27 - December 2, 2011]*, Vol. 8378. Springer, Cham, DEU, 101–136. https://doi.org/10.1007/978-3-319-08915-7_4

[13] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems [Outcome of a Dagstuhl Seminar]*, Vol. 5525. Springer, Cham, DEU, 1–26.

[14] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. 2006. Architecture-Based Self-Adaptation in the Presence of Multiple Objectives. In *Proceedings of the International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS '06)*. ACM, New York, NY, USA, 2–8. https://doi.org/10.1145/1137677.1137679

[15] Ewen Denney, Ganesh Pai, and Josef Pohl. 2012. AdvoCATE: An Assurance Case Automation Toolset. In *Computer Safety, Reliability, and Security (SAFECOMP '12)*, Vol. 7613. Springer, Berlin, DEU, 8–21. https://doi.org/10.1007/978-3-642-33675-1_2

[16] Ewen Denney, Ganesh Pai, and Iain Whiteside. 2017. Model-Driven Development of Safety Architectures. In *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '17)*. IEEE, Piscataway, NJ, USA, 156–166. https://doi.org/10.1109/MODELS.2017.27

[17] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2012. A Formal Approach to Adaptive Software: Continuous Assurance of Non-Functional Requirements. *Formal Asp. Comput.* 24, 2 (2012), 163–186.

[18] Dieter Fox. 2001. KLD-Sampling: Adaptive Particle Filters. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*. MIT Press, Cambridge, MA, USA, 713–720.

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

[20] John Goodenough, Charles Weinstock, and Ari Klein. 2012. *Toward a Theory of Assurance Case Confidence*. Technical Report CMU/SEI-2012-TR-002. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=28067

[21] Michael Grieves and John Vickers. 2017. Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems. In *Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches*. Springer, Cham, DEU, 85–113. https://doi.org/10.1007/978-3-319-38756-7_4

[22] Nico Hochgeschwender, Luca Gherardi, Azamat Shakhirmardanov, Gerhard K. Kraetzschmar, Davide Brugali, and Herman Bruyninckx. 2013. A Model-Based Approach to Software Deployment in Robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '04)*. IEEE, Piscataway, NJ, USA, 3907–3914.

[23] M. Usman Iftikhar and Danny Weyns. 2014. ActivFORMS: Active Formal Models For Self-Adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '09)*. ACM, New York, NY, USA, 125–134.

[24] Daniel Jackson, Martyn Thomas, and Lynette I. Millett (Eds.). 2007. *Software for Dependable Systems: Sufficient Evidence?* National Academy Press, Washington, DC, USA.

[25] Sharmin Jahan, Allen Marshall, and Rose F. Gamble. 2019. Evaluating Security Assurance Case Adaptation. In *Proceedings of the 52nd Hawaii International Conference on System Sciences (HICSS '19)*. ScholarSpace, Manoa, HI, USA, 1–10.

[26] Tim Kelly and Rob Weaver. 2004. The Goal Structuring Notation–A Safety Argument Notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*. CiteseerX, [online], 6.

[27] Jeffrey O. Kephart and David M. Chess. 2003. The Vision Of Autonomic Computing. *Computer* 36 (2003), 41–50.

[28] Jeffrey O. Kephart and Rajarshi Das. 2007. Achieving Self-Management via Utility Functions. *IEEE Internet Computing* 11, 1 (Jan 2007), 40–48. https://doi.org/10.1109/MIC.2007.2

[29] Nathan Koenig and Andrew Howard. 2004. Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '04)*. IEEE, Piscataway, NJ, USA, 2149–2154.

[30] Christos Koulamas and Athanasios Kalogeras. 2018. Cyber-Physical Systems and Digital Twins in the Industrial Internet of Things [Cyber-Physical Systems]. *Computer* 51, 11 (Nov 2018), 95–98. https://doi.org/10.1109/MC.2018.2876181

[31] Jeff Kramer and Jeff Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.* 16, 11 (Nov 1990), 1293–1306.

[32] Zarrin Langari and Tom Maibaum. 2013. Safety Cases: A Review of Challenges. In *Proceedings of the 1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE '13)*. IEEE, Piscataway, NJ, USA, 1–6.

[33] Chung-Ling Lin, Wuwei Shen, Steven Drager, and Betty Cheng. 2018. Measure Confidence of Assurance Cases in Safety-Critical Domains. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, New York, NY, USA, 13–16.

[34] Mike Maksimov, Nick L. S. Fung, Sahar Kokaly, and Marsha Chechik. 2018. Two Decades of Assurance Case Tools: A Survey. In *Computer Safety, Reliability, and Security (SAFECOMP '18)*. Springer, Cham, DEU, 49–59.

[35] Aaron Martinez and Enrique Fernandez. 2013. *Learning ROS for Robotics Programming*. Packt Publishing, Birmingham, UK, Chapter 8. Navigation Stack–Beyond Setups, 266–267.

[36] K. Masaba and A. Q. Li. 2019. ROS-CBT: Communication Benchmarking Tool for the Robot Operating System: Extended Abstract. In *2019 IEEE International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*. IEEE, RUTGERS UNIVERSITY, NEW BRUNSWICK, NJ, USA, 1–3.

[37] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley R. Schmerl. 2015. Proactive Self-Adaptation Under Uncertainty: A Probabilistic Model Checking Approach. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 1–12.

[38] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. 2009. Models@ Run.Time to Support Dynamic Adaptation. *Computer* 42, 10 (Oct 2009), 44–51.

[39] Jason M. O'Kane. 2013. *A Gentle Introduction to ROS*. CreateSpace Independent Publishing Platform, Columbia, South Carolina. http://www.cse.sc.edu/~jokane/agitr/

[40] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. 2009. ROS: An Open-Source Robot Operating System. In *International Conference on Robotics and Automation Workshop on Open Source Software*. IEEE, Piscataway, NJ, USA, 6.

[41] Andres J. Ramirez and Betty H.C. Cheng. 2011. Automatic Derivation of Utility Functions for Monitoring Software Requirements. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS '11)*. Springer-Verlag, Berlin, DEU, 501–516.

[42] Leonardo Ramos, Gabriel Divino, Guilherme Lopes, Breno de França, Leonardo Montecchi, and Esther Colombini. 2019. The RoCS Framework to Support the Development of Autonomous Robots. *Journal of Software Engineering Research*

and Development 7 (2019), 10:1–10:14.

[43] John Rushby. 2015. *The Interpretation and Evaluation of Assurance Cases.* Technical Report SRI-CSL-15-01. Computer Science Laboratory, SRI International, Menlo Park, CA. Available at http://www.csl.sri.com/users/rushby/papers/sri-csl-15-1-assurance-cases.pdf.

[44] The Assurance Case Working Group. 2018. *Goal Structuring Notation Community Standard (Version 2).* Technical Report. SCSC. https://scsc.uk/r141B:1

[45] J. Towler and M. Bries. 2018. ROS-Military: Progress and Promise. In *Ground Vehicle Systems Engineering and Technology Symposium (GVSETS).* National Defense Industrial Association (NDIA), Novi, Michigan, 10.

[46] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. 2004. Utility Functions in Autonomic Systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC '04).* IEEE, Piscataway, NJ, USA, 70–77.

[47] Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. 2012. A Survey of Formal Methods in Self-Adaptive Systems. In *Proceedings of the 5th International C* Conference of Computer Science & Software Engineering (C3S2E '12).* ACM, New York, NY, USA, 67–79.

[48] Danny Weyns, Sam Malek, and Jesper Andersson. 2012. Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems. *ACM Trans. Auton. Adapt. Syst.* 7, 1 (2012), 8:1–8:61.

[49] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. 2009. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *Proceedings of the 17th IEEE International Requirements Engineering Conference.* IEEE Computer Society, Washington, DC, USA, 79–88.

[50] Ji Zhang and Betty H. C. Cheng. 2006. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06).* ACM, New York, NY, USA, 371–380.

[51] Ji Zhang, Heather Goldsby, and Betty H. C. Cheng. 2009. Modular Verification of Dynamically Adaptive Systems. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development.* ACM, New York, NY, USA, 161–172.