Dynamic Assurance of Autonomous Systems through Ground Control Software*

Irfan Sljivo[†], Anastasia Mavridou[‡], Johann Schumann[§], Ivan Perez[¶], Pavlo G. Vlastos[‖], and Corey K. Carter[**]
NASA Ames Research Center, Moffett Field, CA, 94035.

Assurance cases have emerged as a way to build trust in complex autonomous systems. Many assurance case justifications for such systems need to be constantly reevaluated based on the current system context and performance. Autonomous systems, especially those deployed in remote environments, often have a ground control system that enables monitoring and remote operations. In this paper, we propose a dynamic assurance framework that aims at connecting the assurance case with the ground control system. We use the ground control system to facilitate dynamic evaluation of quantitative assurance measures that support various justifications in the assurance case. We demonstrate the proposed dynamic assurance framework on the NASA Ames Research Center project Troupe. We use a combination of in-house and external tools to identify the assurance measures, formalize the related requirements, and generate monitors that feed the data to the external ground control system.

I. Introduction

Assurance cases are being increasingly acknowledged as a way to build trust in complex systems with autonomous capabilities [1]. An assurance case is a comprehensive, defensible, and valid justification that a system will function as intended for the specific mission and operational environment. Such justifications for systems with autonomous capabilities are often based on various quantitative measures of assurance [2]. Due to the dynamic nature of the environmental conditions in which these systems operate, as well as the changing nature of the autonomous systems themselves, these assurance measures cannot be simply estimated once during design time. Rather, they need to be continually evaluated during system operations to ensure that the assurance case justifications still hold. We refer to the assurance case that combines both the static and dynamic elements as a Dynamic Assurance Case (DAC).

Complex systems with autonomous capabilities are often deployed with a Ground Control System (GCS) component to enable remote operation. Whether the system is composed of a single unit or a fleet of units, when deployed in remote environments, the GCS acts as a window into the behavior of the deployed system. The GCS receives telemetry from the system, issues commands to the system and provides various functionalities to visualize the system performance.

We propose a dynamic assurance framework where the GCS keeps track of the assurance measures defined in the system DAC, based on the monitors embedded in the system itself. We first identify different quantitative assurance measures from the system assurance case. Then, we define formal requirements that capture the assurance measures. We generate system monitors needed to evaluate the assurance measures with the help of the formalized requirements. Finally, we use the GCS to evaluate and visualize these assurance measures, which allows us to continuously evaluate the dynamic assurance case justifications that rely on these measures.

We demonstrate our dynamic assurance framework in the NASA Ames Research Center project Troupe, which aims at developing a fleet of rovers capable of autonomously mapping their environment. The rovers work cooperatively, each collecting data for different parts of the environment. Each rover runs an identical core Flight System (cFS) [3] application. Troupe uses OpenC3 Cosmos [4] as the GCS, and AdvoCATE [5] to capture the system DAC. We use

*GOVERNMENT RIGHTS NOTICE. This work was authored by employees of KBR Wyle Services, LLC under Contract No. 80ARC020D0010 with the National Aeronautics and Space Administration. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, or allow others to do so, for United States Government purposes. All other rights are reserved by the copyright owner.

[†]Intelligent Systems Division, Robust Software Engineering, NASA Ames Research Center/KBR.
[‡]Intelligent Systems Division, Robust Software Engineering, NASA Ames Research Center/KBR.
[§]Intelligent Systems Division, Robust Software Engineering, NASA Ames Research Center/KBR.
[¶]Intelligent Systems Division, Robust Software Engineering, NASA Ames Research Center/KBR.
[‖]Intelligent Systems Division, Robust Software Engineering, NASA Ames Research Center/KBR.
[**]Intelligent Systems Division, Robust Software Engineering, NASA Ames Research Center.

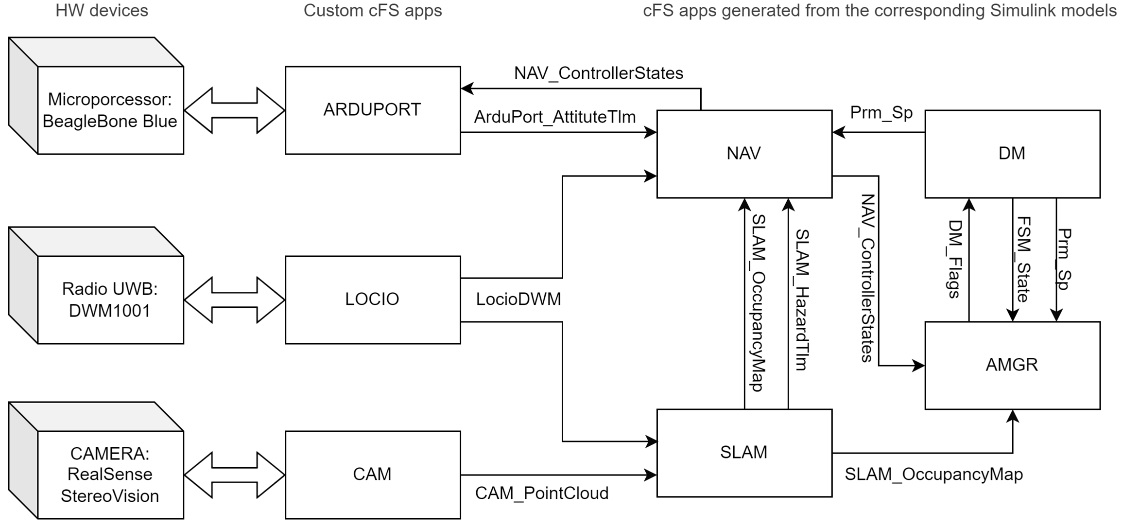

Fig. 1 High-level architecture of the Troupe user-level application layer.

the methodology for integrating the assurance case with formal verification [6] to connect the assurance case with formal-based runtime monitoring tools. In particular, we use FRET [7, 8] to formalize the requirements captured in AdvoCATE. Finally, to generate cFS monitors and obtain system information needed for evaluating the system DAC, we leverage the capabilities of two runtime monitoring tools: 1) Ogma/Copilot [9–11] and 2) R2U2 [12].

II. Troupe Overview

Troupe aims at developing a fleet of rovers capable of autonomously mapping their environment. The rovers work cooperatively, each collecting data for different parts of the environment, and building an occupancy map of its environment, which it shares with other rovers. The Troupe system software is built on top of the NASA's core Flight System (cFS) middleware. cFS consists of a set of reusable components or *applications* that communicate with each other via a software bus. Examples of such applications include Telemetry Output (TO), for sending telemetry packets to a remote address, Command Ingest (CI), for receiving commands from a remote address, and Scheduler (SCH), for generating software bus messages at pre-determined timing intervals. On top of these reusable cFS components, we define the user cFS applications such as SLAM, CAM and LOCIO; where LOCIO determines the current position of the rover, and SLAM builds the occupancy map based on the images captured with CAM. Fig. 1 shows the high-level architecture of the user-level applications. The connections between the applications represent the packets shared on the software bus to which the other applications are subscribed to. For example, LOCIO publishes the packet named LocioDWM to the software bus, and both NAV and SLAM applications use the information provided in that packet. On top of the user-level application layer, we add two additional applications for runtime monitoring R2U2 and Copilot. Full overview of Troupe is presented in the Troupe system paper [13].

III. Dynamic Assurance Case Methodology for cFS

AdvoCATE (Assurance Case Automation Toolset [5]) is a tool that supports the development and management of assurance cases. A safety assurance case comprises all the artifacts that are created during system development and verification that are needed to assure that the system is acceptably safe for its intended operation. Safety cases are often represented and documented in the form of a graphical argument that presents how the system safety goals have been achieved and are supported by the various items of *evidence*, such as test results, simulations, and formal verifications. AdvoCATE supports a range of notations and modeling formalisms, including Goal Structuring Notation (GSN) [14] to document safety cases and Bow-Tie Diagrams (BTD) [15] for risk modelling. To enable automation of the development and management of assurance cases, AdvoCATE implements an assurance metamodel that allows all of the artifacts relevant from the safety assurance perspective to be explicitly defined and their relationships captured. Some of the artifacts can be created directly in AdvoCATE (e.g., hazard log, safety arguments), while other artifacts, such as formal

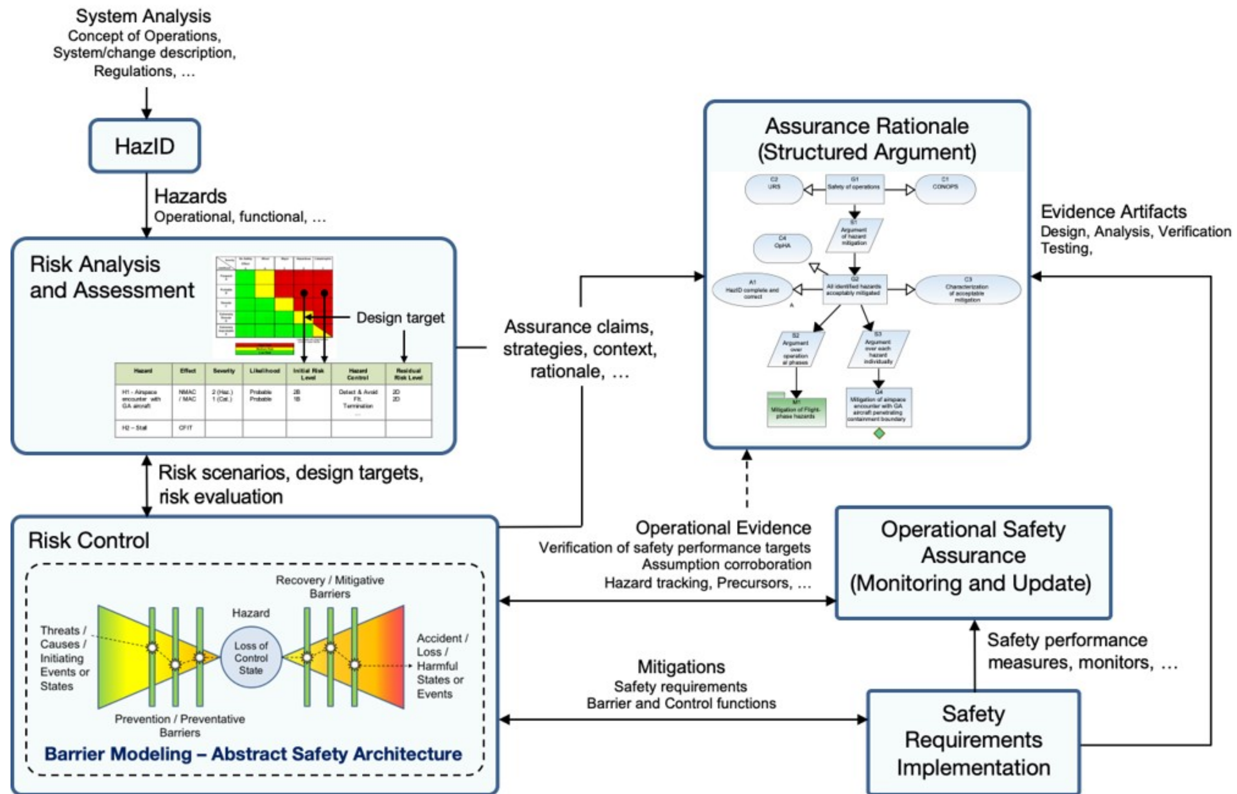

Fig. 2 Overview of the AdvoCATE assurance methodology [16]

verification results, can be imported into the tool so that the evidence can be collectively viewed.

The high-level AdvoCATE assurance methodology is presented in Fig. 2. The rounded rectangles represent the different activities, and arrows the data flow. Briefly, the assurance methodology starts by defining the system in terms of physical and functional decompositions with failure modes and functional deviations, respectively. Then, hazard analysis is performed based on the system definition and recorded in the *hazard log*. Each hazard, along with its causes, consequences, and their mitigations, is depicted in a *bow-tie diagram* that is used for risk modelling and control. An example BTD is shown in Fig. 3. The composition of all bow tie diagrams comprises the *safety architecture* of the system. Information from the safety architecture and evidence artefacts is used to construct structured arguments presenting the system assurance rationale.

In this paper we focus on detailing the Operational Safety Assurance also referred to as Dynamic Safety Assurance. We follow these steps to connect a system with its dynamic assurance case through the ground control software:

- Assurance measures definition in AdvoCATE.
- Requirement formalization in FRET.
- Generation of monitors as cFS applications using Ogma/Copilot and R2U2.
- Assurance measure monitoring in Cosmos.

In the remainder of this section we detail each of these steps and demonstrate on a part of the Troupe system.

A. Assurance Measure Definition

To define assurance measures, we assume that a partial safety architecture for the system is developed. Controls and events are the building blocks of the safety architecture. Each event is associated with the initial and residual risk estimate. The residual values are calculated based on the event connections and the integrity of the applied controls. For example, if a control is completely mitigating an event (i.e., it has a high control integrity), then the likelihood of its consequence decreases, which lowers the residual risk of the consequence.

We define the assurance measures based on the effects that certain events and controls have on the residual risk estimate of the top event in the safety architecture. For example, hazardous events regarding the self position function in

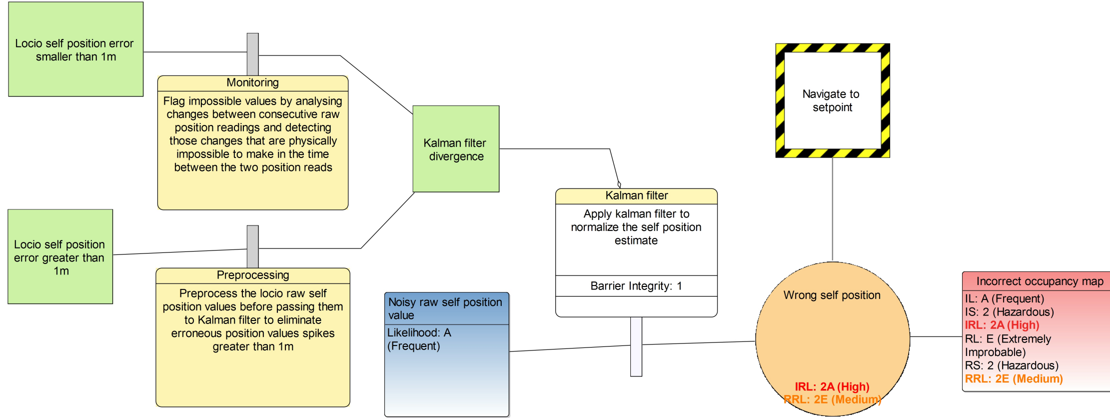

Fig. 3 Bow-Tie Diagram of the wrong self position hazard (orange circle), its causes (blue rectangles to the left), controls applied to mitigate the causes (the Kalman filter box), escalation factors jeopardizing the controls (green rectangles to the left), and the controls over the escalation factors (yellow boxes).

Table 1 Mitigation requirements specified for some of the mitigation measures from the safety architecture

L5-LOC-001	Runtime monitor shall raise an invalid location flag if invalid location changes are detected.
L5-LOC-002	Locio shall monitor for impossible value changes and raise an invalid location flag.
L5-LOC-003	Locio shall preprocess the raw positions before publishing the LocioDWM and discard out-of-bound values.
L5-LOC-004	Runtime monitor shall raise a flag if LOCIO position quality factor remains critical and does not become low for more than 1 seconds.
L5-LOC-005	Locio position average shall include only position values that have the invalid location flag set to false.

Troupe directly influence the resulting residual risk of whether a rover may collide with an obstacle or stay stuck in a place waiting for the self position to have the required integrity level. If the estimates of the likelihood of the raw position value fails or the integrity of controls that aim at those failures from propagating change, the resulting residual risk of the top events changes as well. We define assurance measures by formalizing the components of risk of the influential events and the integrity of influential controls. Examples of assurance measures include:

- Number of times self position error has been detected
- Number of times the performance of the rover was degraded because of the incorrect current position
- The trend of self position error during the last x seconds

Fig. 3 shows a BTD for a single hazard in the system regarding incorrect current position estimate. Since the position sensor is noisy, we apply a Kalman filter to produce a more accurate estimate of the self position. However, even the Kalman filter may be compromised and diverge if large value failures happen. We have observed two kinds of large value failures here, the ones that are clearly detectable by comparing the position values to a predefined threshold, and the kind of failures that need more elaborate identification based on the system and environment constraints, e.g., the speed of the rover, heading, movement pattern. To address these two events that may compromise the Kalman filter, we define separate mitigation mechanisms to minimize their effect on the Kalman filter. The residual risk level of the Wrong Self position and its consequence completely depend on these three controls. We define the assurance measure in terms of the integrity of the controls that aim at reducing the risk of the Kalman filter divergence. The initial integrity estimate for both controls was that they will catch 99% of the large value failures and prevent them from propagating to the Kalman filter. The requirements for some of the mitigation measures are shown in the Table 1.

B. Requirement Formalization

We specify requirements using FRET (Formal Requirements Elicitation Tool [7]), developed at the NASA Ames Research Center. FRET is an open-source tool [17] for writing, understanding, formalizing, and analyzing requirements. In practice, requirements are typically written in natural language, which is ambiguous and, consequently, not amenable to formal analysis. Since formal, mathematical notations are unintuitive, requirements in FRET are entered in a restricted

Fig. 4 FRET Requirement for invalid location.

natural language named FRETish [18] with precise, unambiguous meaning.

Let us look at two Troupe requirements and how we specified these in FRETish. Figures 4 and 5 present two distinct requirement examples specified in the '*Update Requirement*' pane of FRET. The pane is split into two parts - the editor on the left and the assistant tab on the right.

Each requirement has a a unique name written in the '*Requirement ID*' field of the editor. Both example requirements are L5 requirements that belong to the LOCIO component of the Troupe system. Requirements in FRET are organized in projects (see '*Project*' field). Free text can be optionally entered in '*Rationale and Comments*'.

'*Requirement Description*' shows the requirement written in FRETish. Each FRETish requirement is composed using up to six distinct fields. We use the example of Fig. 5 to explain the FRETish fields. This requirement checks the position quality factor of LOCIO for which we consider different levels of criticality: "critical" (pos-qf-critical), "low" (pos-qf-low), and "good" (pos-qf-good). We require that if the level is critical then it must return back to low level within 1 second.

Next, we explain the six FRETish fields (the * symbol designates mandatory fields): 1) `scope` specifies the time intervals where the requirement is enforced - if left empty, scope is the complete execution trace, 2) `conditions` (**whenever pos_df_critical**) is a Boolean expression that whenever true specifies that the `responses` shall happen 3) `component*` (**LOCIO**) is the system component that the requirement is levied upon, 4) `shall*` is used to express that the component's behavior must conform to the requirement, 5) `timing` (**within 1 seconds**) specifies when the response shall happen, subject to the constraints defined in `scope` and `conditions` and 6) `responses*` (**satisfy pos_qf_low**) is the Boolean expression that the component's behavior must satisfy.

Once the FRETish requirement parses successfully, the user may click on the '*SEMANTICS*' button to generate English and diagrammatic explanations of the requirement semantics in the '*ASSISTANT*' tab. Additionally, FRET

Fig. 5 FRET Requirement for LOCIO position quality factor.

automatically produces formalizations in Future Time LTL and Past Time LTL (in SMV and Lustre formats). Since getting a requirement with temporal relationships right can be a tricky and challenging task, FRET provides an interactive requirements visualizer, available by clicking '*SIMULATE*' in the assistant tab (see Fig. 5). Given a FRET requirement, the simulator shows temporal traces of each of the signals (variables) involved as well as the valuation of the requirement for each point in time. The user may interactively modify the input signals, and then the valuation of the requirement is updated automatically, i.e., it becomes green if the requirement is satisfied or red if the requirement is violated. Figures 6 and 7 show different valuations of the LOCIO position critical factor requirement. In both Figures 6 and 7, the first temporal trace corresponds to variable `pos_qf_critical`, the second to variable `pos_qf_low` and the last trace corresponds to the valuation of the complete requirement `L5-LOC-004`.

Through its analysis portal, FRET connects to analysis tools by facilitating the mapping between requirements and models/code, and by generating verification code [19–22]. In this paper, we show how formalized requirements can be used for creating monitors to be run by the R2U2 and Copilot tools.

C. Generation of Safety Monitors for cFS

1. The Ogma/Copilot process

Ogma [9, 10] is a monitoring application generation tool developed in collaboration between NASA Langley Research Center and NASA Ames Research Center. Among other features, Ogma is capable of generating cFS applications that gather information from multiple other cFS applications via the software bus, re-evaluate values being calculated every time that new data arrives in the software bus, and relay the results to other applications or to a ground

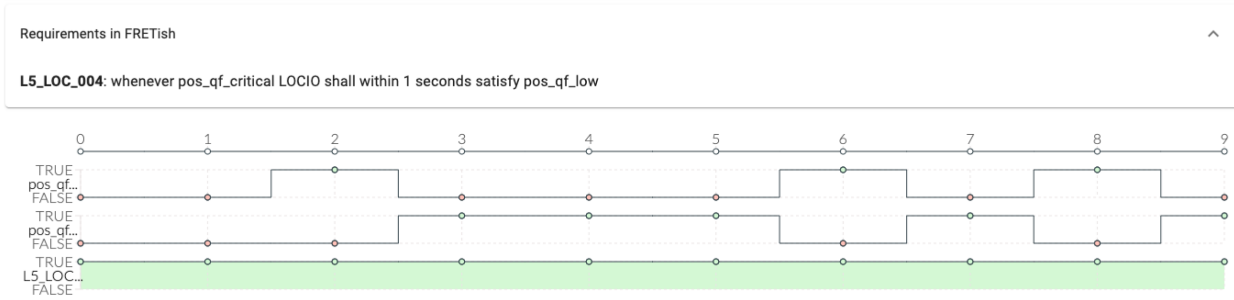

Fig. 6 FRET Simulator shows an execution trace where requirement L5_LOC_004 is satisfied.

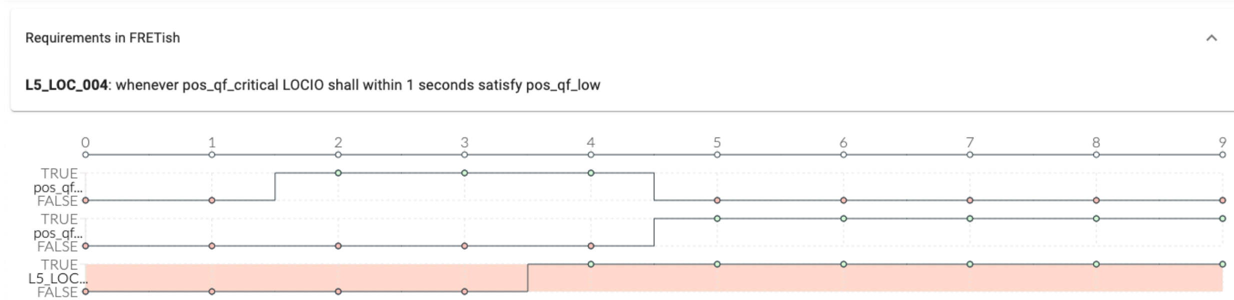

Fig. 7 FRET Simulator shows an execution trace where requirement L5_LOC_004 is violated.

station (Fig. 8). *

To implement the core of the monitors, Ogma relies on Copilot [11], a high-level runtime monitoring language developed by the same team. From the Copilot specifications, the compiler produces real-time C99 code that implements the monitors.† In all cases, the monitors produced by Copilot execute in predictable time and with bounded memory, making them suitable for embedded systems with limited resources. The use of C99 as the target language also ensures maximum portability. Apart from the C99 code, the Copilot compiler comes with a verifier [23] that produces a formal proof of correctness of the code generated. The proof establishes that the C code behaves as expected based on the original specification. The C99 code produced is compliant with MISRA C.‡

From the assurance case, we can determine what information needs to be monitored, create the specifications needed, and use Ogma and Copilot to generate the necessary application code that executes the monitors. There are mainly two pieces of information that we need to provide to these tools to generate the monitoring applications. The first one is a list of the data that must be monitored. It can consist of data that is being published in the software bus by other cFS applications, requirements or properties that must be monitored, and data that can be synthesized from either, such as a counter of the frequency with which a property is being violated. Depending on its nature, this data specification can be given directly in the Copilot language, as well as in other high-level languages like Lustre [24] and FRET [19, 25]. The second is a plain-text database that maps data by name to the cFS message that carries such data. Such a database only needs to be defined once per project. From these two pieces of information, Ogma generates hard-realtime code that processes the data to be monitored as new samples arrive. The monitoring code is wrapped in a cFS application that subscribes to the messages providing information used by those monitors, re-evaluates the monitors as needed, and publishes the results. The applications generated by these tools are meant to be built together with other cFS applications and require no modifications after being generated. In our architecture, such results are then obtained through the Telemetry Output (TO) application, which can relay them to the Ground Station for further processing.

*Other targets supported by Ogma include the Robot Operating System (ROS 2), and FPrime.

†The Copilot compiler can also target FPGAs; in this work, we used only the C99 backend.

‡As of the time of this writing, the code produced complies with all rules in MISRA C 2012, and all but one directives.

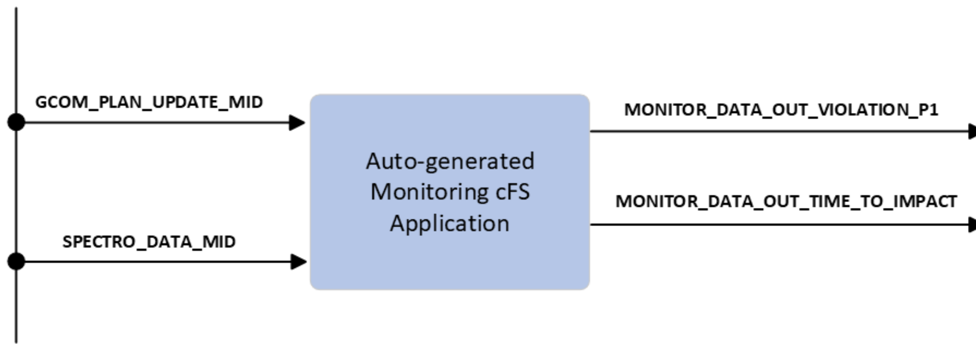

Fig. 8 High-level view of cFS monitoring application generated by Ogma.

2. The R2U2 process

R2U2 (Realizable, Responsive, Unobtrusive Unit) [26] is a framework and tool for the continuous monitoring of safety-critical and embedded cyber-physical systems. R2U2 combines past-time and future-time Metric Temporal Logic, probabilistic reasoning with Bayesian networks, and model-based prognostics.

Like the other components of Troupe, R2U2 is implemented as a cFS app and activated at a regular rate of 1Hz. Fig. 9 shows its architecture. R2U2 subscribes to numerous messages of the software bus that carry sensor data and current rover status. Using a set of customizable filters, operators, and discretizers, R2U2 produces Boolean values, which are then processed by the R2U2 temporal engine [27, 28].

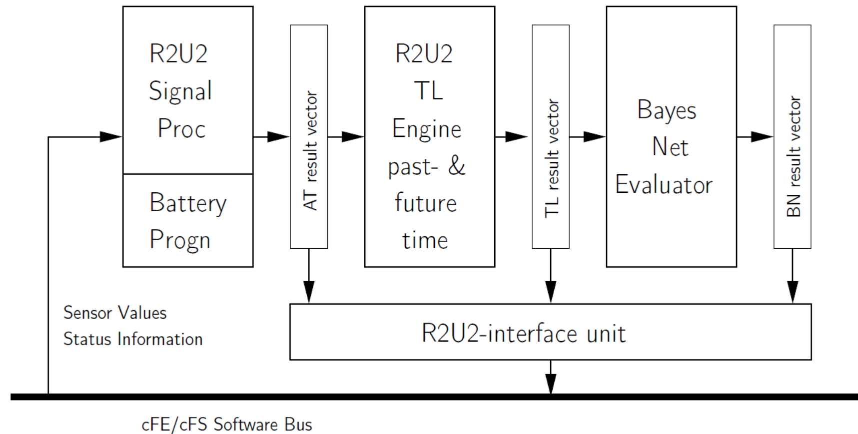

Fig. 9 The R2U2 monitoring system.

Because R2U2 is implemented as a separate app, it is capable of monitoring the overall troupe system without requiring code instrumentation or modifications to the rover control code or individual applications. With this *unobtrusive* architecture, it can be assured that the system behavior is not affected by runtime monitoring.

Since in the current architecture, R2U2 is only invoked at a low 1Hz rate, the additional burden on the overall system is kept minimal. An R2U2 implementation on a co-processor [29] could even further minimize its footprint.

Requirements as captured by FRET are then automatically translated into temporal logic (as explained in Section III.B) and used as formulas for R2U2's temporal engine. Typically, those requirements can be grouped to:

- SW-related requirements are specific properties and behaviors of the troupe SW system to be monitored. E.g., the rate, with which the LOCIO app publishes a certain message, can be monitored. Also state transitions and conditions of the DM and their correctness and consistency can be checked.
- Sensor-related requirements are requirements about the quality and validity of sensors. For example, the minimal frame-rate of a camera, or the quality of the UWB sensor network are typical examples. Such requirements/properties are often used to detect failures and anomalies.

- Behavioral properties are used to monitor safe behavior and performance of the rovers. A typical example might include: if a move forward command has been issued, then within a short time-frame, a positive (forward) speed must be detected. In a more detailed property, failure of detecting that speed can be diagnosed as (a) problems with motor(s) by monitoring motor current, or (b) a situation where the rover is stuck in a dust bowl (motor and wheels turning, but no movement).

The basis for the monitoring properties are requirements, defined in FRETish specifications and properties, given in the R2U2 input language, as well as Bayesian networks (not used in this paper). Figure 9 shows, how the given given properties, specifications, and Bayesian Networks are, during compile time, automatically converted into efficient machine-readable formats and how the compiled C-code is linked with the skeleton of the R2U2 app. The C code of the app or the system-wide configuration definitions only need to be touched if R2U2 needs to monitor additional message types of cFS SW bus messages. R2U2 publishes its results as a specific message with ID `R2U2_DATA_OUT_MID`. These messages can be transmitted to the ground station for dynamic assurance and can be used by other on board components, e.g., the DM, as basis for decision making and contingency planning [30].

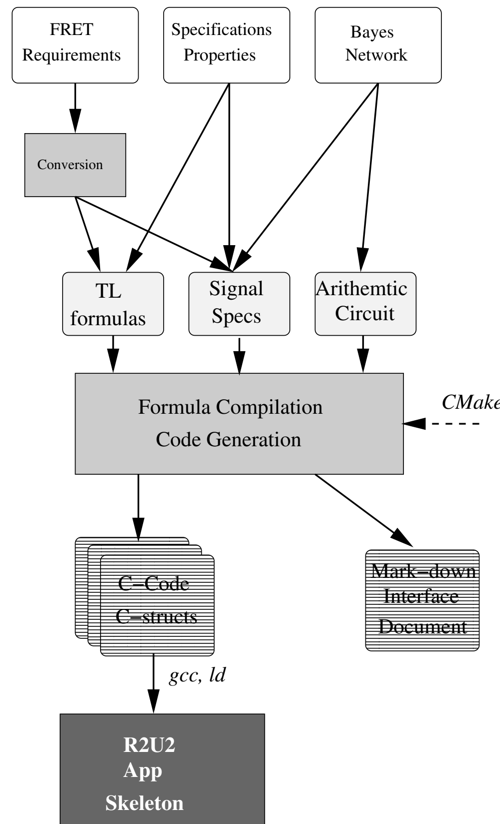

Fig. 10 Process for customization and building the R2U2 app.

D. Monitoring Experiments

Fig. 11 shows relevant signals of running a scenario test with one rover equipped with Copilot and R2U2. The top 3 panels show the UWB position estimates and the position quality metric (qf). Green lines indicate the raw measurements, blue lines measurements running through an R2U2 low-pass filter of length 4, and red lines comprise the position average over the last second, as provided by the UWB firmware. Panel 4 and 5 show the speed of the rover in x and y direction; these values are obtained from R2U2, which applies a rate filter to the position signals.

In our experiment, the rover was kept stationary on a table for the first 60 seconds. Then the rover was moved a bit (still on the table) for the next 30 seconds. After that the rover was moved from the table and back again (seconds 90–110). At around t=130s the rover was moved quickly away from the table and put back at t=150 where it stayed

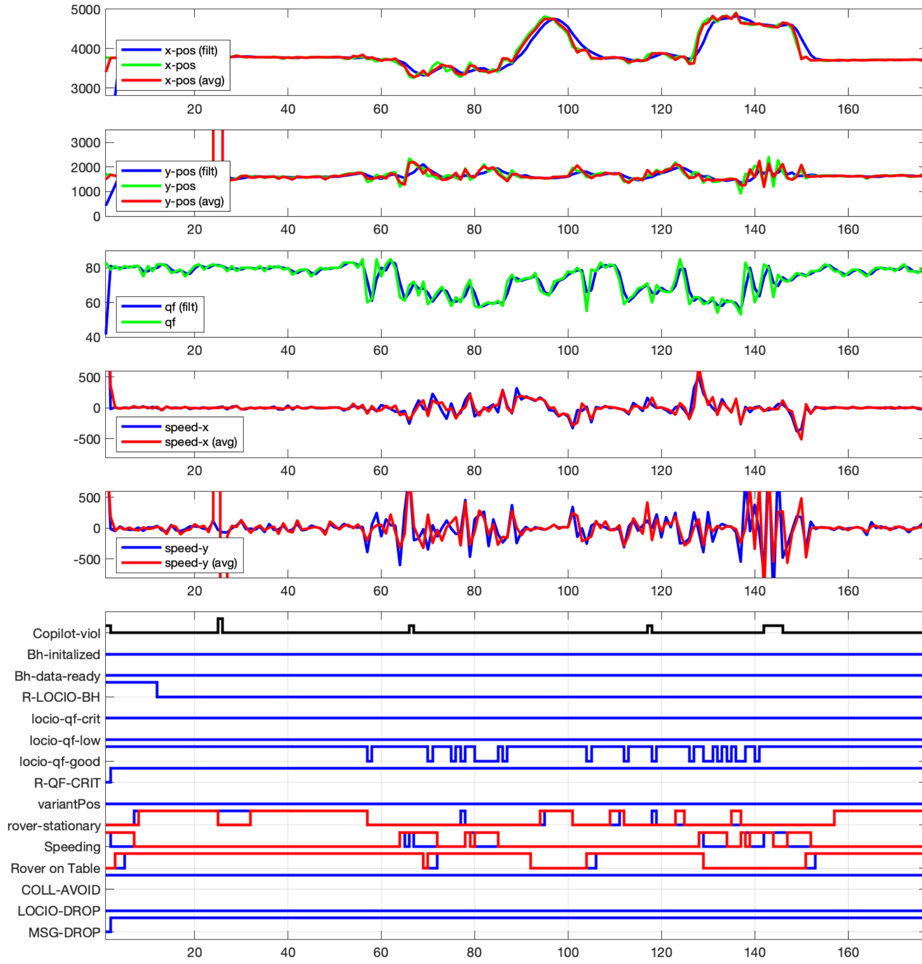

Fig. 11 RU2U and Copilot experiments.

stationary until the end of the experiment.

The bottom panel shows signals as generated by Copilot and R2U2. The first line shows the number of violations of the property shown in Fig. 4 from the generated Copilot monitor. As expected it coincides with unreasonable position and speed estimates in the y-direction. In our example, R2U2 properties focus on UWB quality and behavior of the rover: the quality of the estimates never went into critical or even low regimes. However, at certain times, when the rover was moved, it dropped below the optimal threshold of 70%. This behavior resulted that our quality requirement shown in Fig. 5 was never violated (R-QF-CRIT) except during startup time.

Based on the estimated speed, R2U2 monitored if the rover was stationary or moving. The blue line corresponds to $H[0,5](speed < \theta)$, the red line shows that information based upon the "average" signal provided by UWB. That signal tends to ignore shorter (noisy) false alarms but is plagued by the signal dropouts as reported by Copilot.

"Speeding" occurs whenever the speed in x or y direction exceeds a specific threshold. The raw data are shown in blue, the red line corresponds to "have not speeded in the last 5 seconds": $O[0,5](\neg(rate_x < \theta) \vee \neg(rate_y < \theta))$.

Finally, the "Rover on Table" line indicates when the rover is on the table, based upon the UWB position estimated. Here again the red line is a temporal logic filter to eliminate short false alarms.

Fig. 12 shows some of the properties from Fig. 11 captured in the live telemetry screens in OpenC3 Cosmos GCS.

Fig. 12 Some of the monitored properties as shown in the live telemetry screens in OpenC3 Cosmos

While the data for Fig. 11 is extracted for a specific scenario to better showcase the different properties, Fig. 12 shows another instance of the experiment captured directly in GCS.

IV. Conclusions

Dynamic assurance of various assurance case justifications is needed both during the design and operations of an autonomous system. We have proposed a dynamic assurance framework where we use the safety architecture of the system developed in AdvoCATE to identify various quantitative assurance measures that need to be continuously evaluated to monitor their effects on the residual risk in the system. We used FRET to specify and formalize the mitigation requirements of the controls from the safety architecture. Then, we used Ogma/Copilot and R2U2 to generate monitors corresponding to the formalized requirements. Finally, we used the OpenC3 Cosmos ground control system to read telemetry and visualize assurance measures. We demonstrated the proposed dynamic assurance methodology on the NASA Ames Research Center project Troupe, that aims at developing a rover swarm tasked with autonomously exploring an unknown terrain.

In the future, we plan to explore support for connecting the assurance case in AdvoCATE with the ground control systems such as OpenC3 Cosmos, so that the assurance measure data can be directly embedded in the assurance case itself. Alternatively, we could generate screens from the assurance case tool that could be used to monitor the assurance measures in the ground control system. Furthermore, we plan to investigate the different kinds of assurance measures that can be identified from the system safety architecture, and all the points in the assurance case where the corresponding assurance measure evaluations could be used. Finally, we plan to further improve the automated process of generating runtime monitors in R2U2 and Copilot from the formalized requirements in FRET.

References

[1] Asaadi, E., Denney, E., Menzies, J., Pai, G. J., and Petroff, D., "Dynamic Assurance Cases: A Pathway to Trusted Autonomy," *Computer*, Vol. 53, No. 12, 2020, pp. 35–46. https://doi.org/10.1109/MC.2020.3022030.

[2] Asaadi, E., Denney, E., and Pai, G., "Towards quantification of assurance for learning-enabled components," *2019 15th European Dependable Computing Conference (EDCC)*, IEEE, 2019, pp. 55–62.

[3] McComas, D., "NASA/GSFC's Flight Software Core Flight System," *Flight Software Workshop*, Vol. 11, 2012.

[4] OpenC3, https://openc3.com/, Accessed: 2023-06-01.

[5] Denney, E., and Pai, G., "Tool Support for Assurance Case Development," *Automated Software Engineering*, Vol. 25, No. 3, 2018, pp. 435–499.

[6] Bourbouh, H., Farrell, M., Mavridou, A., Sljivo, I., Brat, G., Dennis, L. A., and Fisher, M., "Integrating formal verification and assurance: an inspection rover case study," *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*, Springer, 2021, pp. 53–71.

[7] Giannakopoulou, D., Mavridou, A., Pressburger, T., Rhein, J., Schumann, J., and Shi, N., "Formal Requirements Elicitation with FRET," *REFSQ*, 2020.

[8] "FRET: Formal Requirements Elicitation Tool," , Accessed: 2023. URL https://github.com/NASA-SW-VnV/fret.

[9] Ogma, https://github.com/nasa/ogma, Accessed: 2023-06-01.

[10] Dutle, A., Muñoz, C., Conrad, E., Goodloe, A., Perez, I., Balachandran, S., Giannakopoulou, D., Mavridou, A., Pressburger, T., et al., "From requirements to autonomous flight: an overview of the monitoring ICAROUS project," *arXiv preprint arXiv:2012.03745*, 2020.

[11] Perez, I., Dedden, F., and Goodloe, A., "Copilot 3," Tech. Rep. NASA/TM-2020-220587, NASA Langley Research Center, April 2020.

[12] Rozier, K. Y., and Schumann, J., "R2U2: Tool Overview," *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, 2017, pp. 138–156. URL http://www.easychair.org/publications/paper/Vncw.

[13] Benz, N. A., Sljivo, I., Woodard, A., Vlastos, P. G., Carter, C. K., and Hejase, M., "The Troupe System: An Autonomous Multi-Agent Rover Swarm," *AIAA SciTech 2024*, 2024.

[14] "GSN Community Standard Version 3," Tech. rep., Assurance Case Working Group of The Safety-Critical Systems Club, May 2021. URL https://scsc.uk/r141C:1.

[15] Authority, C. A., "Bowtie risk assessment models," , 2019.

[16] Denney, E., Pai, G., and Whiteside, I., "The role of safety architectures in aviation safety cases," *Reliability Engineering & System Safety*, Vol. 191, 2019, p. 106502.

[17] FRET, https://github.com/NASA-SW-VnV/fret/, Accessed: 2023-06-01.

[18] Giannakopoulou, D., Pressburger, T., Mavridou, A., and Schumann, J., "Automated formalization of structured natural language requirements," *Information and Software Technology*, Vol. 137, 2021, p. 106590. https://doi.org/10.1016/j.infsof.2021.106590.

[19] Perez, I., Mavridou, A., Pressburger, T., Goodloe, A., and Giannakopoulou, D., "Automated Translation of Natural Language Requirements to Runtime Monitors," *Tools and Algorithms for the Construction and Analysis of Systems*, edited by D. Fisman and G. Rosu, Springer International Publishing, Cham, 2022, pp. 387–395.

[20] Mavridou, A., Bourbouh, H., Garoche, P. L., Giannakopoulou, D., Pessburger, T., and Schumann, J., "Bridging the Gap Between Requirements and Simulink Model Analysis," *Joint 26th International Conference on Requirements Engineering: Foundation for Software Quality Workshops, Doctoral Symposium, Live Studies Track, and Poster Track*, 2020.

[21] Ferro, C. M. d., Mavridou, A., Dille, M., and Martins, F., "Simplifying Requirements Formalization for Resource-Constrained Mission-Critical Software," *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2023, pp. 263–266. https://doi.org/10.1109/DSN-W58399.2023.00066.

[22] Ádám, Z., Lopez-Miguel, I. D., Mavridou, A., Pressburger, T., Bęś, M., Blanco Viñuela, E., Katis, A., Tournier, J.-C., Trinh, K. V., and Fernández Adiego, B., "From Natural Language Requirements to the Verification of Programmable Logic Controllers: Integrating FRET into PLCverif," *NASA Formal Methods*, edited by K. Y. Rozier and S. Chaudhuri, Springer Nature Switzerland, Cham, 2023, pp. 353–360.

[23] Scott, R. G., Dodds, M., Perez, I., Goodloe, A. E., and Dockins, R., "Trustworthy Runtime Verification via Bisimulation (Experience Report)," *Proc. ACM Program. Lang.*, Vol. 7, No. ICFP, 2023. https://doi.org/10.1145/3607841, URL https://doi.org/10.1145/3607841.

[24] Gacek, A., Backes, J., Whalen, M., Wagner, L., and Ghassabani, E., "The JKind model checker," *International Conference on Computer Aided Verification*, Springer, 2018, pp. 20–27.

[25] Perez, I., Mavridou, A., Pressburger, T., Will, A., and Martin, P. J., "Monitoring ROS2: from Requirements to Autonomous Robots," *Formal Methods for Autonomous Systems Workshop*, 2022.

[26] Reinbacher, T., Rozier, K. Y., and Schumann, J., "Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems," *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science (LNCS), Vol. 8413, Springer, 2014, pp. 357–372.

[27] Geist, J., Rozier, K. Y., and Schumann, J., "Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems," *RV14*, 2014, pp. 215–230.

[28] Schumann, J., Roychoudhury, I., and Kulkarni, C., "Diagnostic Reasoning using Prognostic Information for Unmanned Aerial Systems," *Proceedings of the 2015 Annual Conference of the Prognostics and Health Management Society (PHM2015)*, 2015.

[29] Schumann, J., and Moosbrugger, P., "Unobtrusive Software and System Health Management with R2U2 on a parallel MIMD Coprocessor," *Proceedings of the 2017 Annual Conference of the Prognostics and Health Management Society (PHM2017)*, 2017.

[30] Hejase, M., Katis, A., and Mavridou, A., "Design, Formalization, and Verification of Decision Making for Intelligent Systems," AIAA/Scitech, 2024.