

Dynamic Assurance of Autonomous Systems through Ground Control Software

Irfan Sljivo*, Ivan Perez[†] and Anastasia Mavridou[‡]
KBR/NASA Ames Research Center, Moffett Field, CA, 94035.

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 operating environment. Such justifications for systems with autonomous capabilities are often based on various probabilistic quantifications [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 probabilistic quantifications cannot be simply estimated once during design time. Rather, they need to be continually evaluated during systems operations to ensure that the assurance case justifications are valid. We refer to the assurance case that combines both the static and dynamic elements as a Dynamic Assurance Case (DAC).

Such complex systems with autonomous capabilities are often deployed with a Ground Control Software (GCS) component to enable remote operation. Whether the system is composed of a single unit or a fleet of units, deployed distributed or in remote environments, GCS acts as a window into the behavior of the deployed system. It 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 acts as a relay between the autonomous system and its DAC. GCS itself can be used to track both unit-specific as well as system-wide probabilistic quantifications using the incoming telemetry. We embed these quantifications throughout the DAC as variables that can be updated by external sources. We use the GCS to periodically update these variables, which allows us to continually evaluate the formally defined assurance case justifications.

We demonstrate our dynamic assurance framework in the NASA Ames project Troupe that 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 ground system, and AdvoCATE [5] to capture the system DAC. We use 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 the FRET tool [7] to formalize the requirements captured in AdvoCATE. Then, we leverage the capabilities of the Ogma [8, 9] and Copilot [10] tools, and their integration with FRET, to generate cFS monitors and obtain system information needed for updating the system DAC. We show how we can capture both rover-specific and system-wide quantifications in Cosmos using its Ruby scripting editor and pass them into the DAC modelled in AdvoCATE. Then, we show how these incoming variables can be embedded in different parts of the DAC and how effects of their updates can be observed.

II. 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 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) [11] to document safety cases and Bow-Tie Diagrams (BTD) [12] for risk modelling. To enable automation of the development

*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.

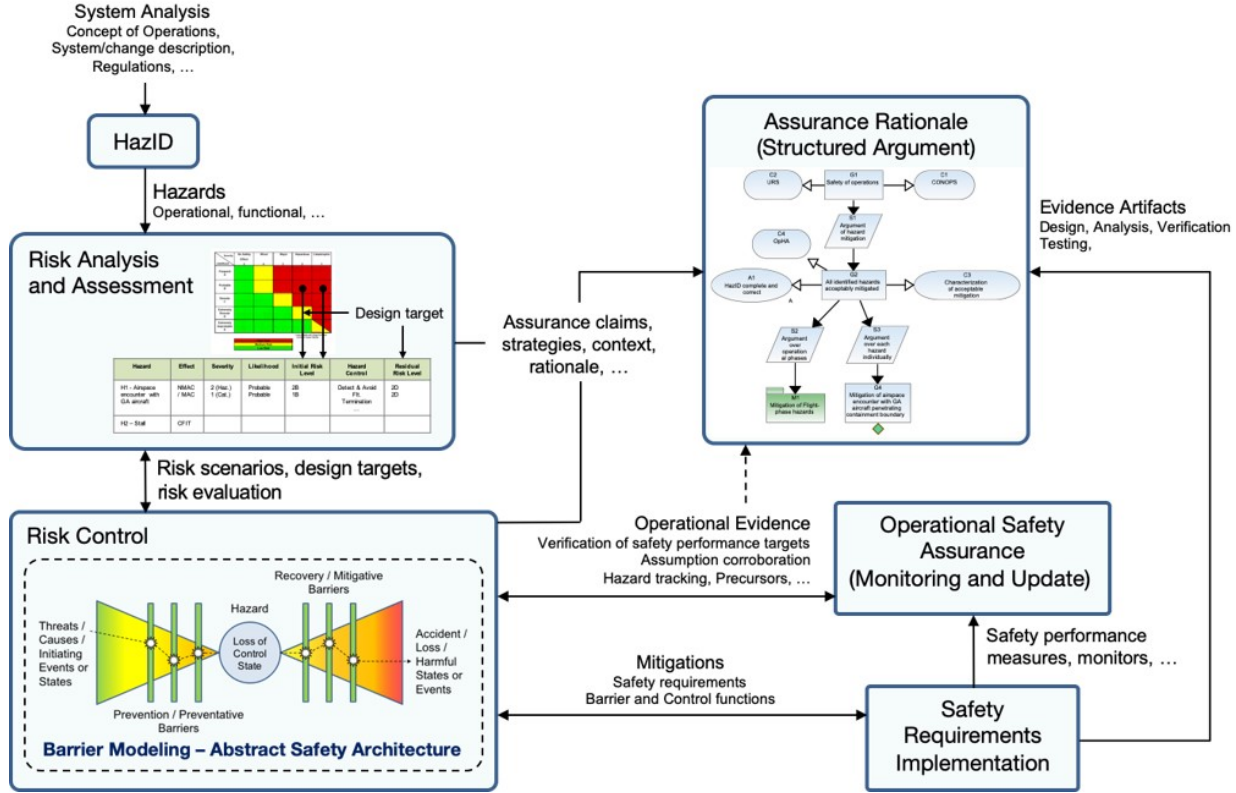


Fig. 1 Overview of the AdvoCATE assurance methodology [13]

and management of assurance cases, AdvoCATE implements an assurance metamodel that allows all of the artifacts relevant from the safety assurance perspective can be explicitly defined and their relations captured. Some of the artifacts can be created directly in AdvoCATE (e.g., hazard log, safety arguments), while other artifacts such as formal 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 the Figure 1. 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 BTM is shown in Figure 2. 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 cFS system with its dynamic assurance case through the ground control software:

- Safety performance metric definition in AdvoCATE
- Requirement formalization in FRET
- Generation of safety performance metric monitors as cFS apps using Ogma and Copilot

In the reminder of this section we briefly detail each of these steps and give examples from the Troupe system. Full Troupe system description is given in a companion paper submitted to AIAA SciTech, and would be referred to here, and parts of it would be included to make the paper self contained.

A. Safety performance metric definition

To define safety performance metrics, we assume that a partial safety architecture for the system is developed. Figure 2 shows a BTM for a single hazard in the system regarding incorrect current position estimate. There are multiple causes that can lead to incorrect position, including stuck at value failures, large value failures, HW link disruptions

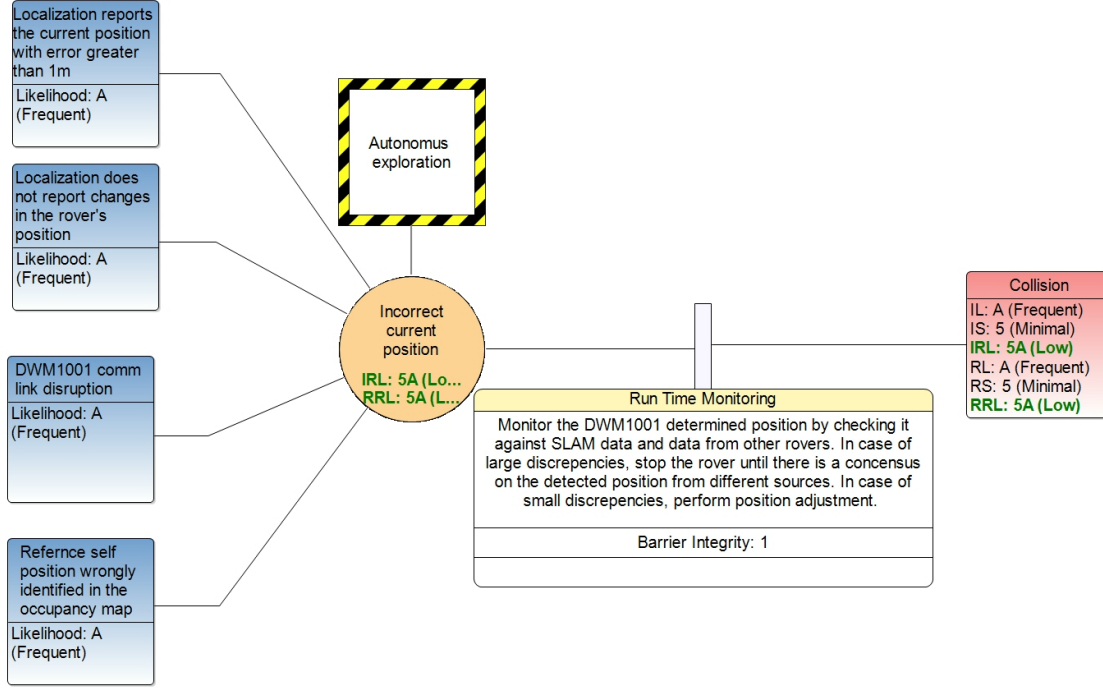


Fig. 2 Bow-Tie Diagram of the incorrect current position hazard (orange circle), its causes (blue rectangles to the left), its consequence (red rectangle to the right), and the control applied to mitigate the hazard (Run Time Monitoring box).

etc. We note some of the failures in the BTd. DWM1001 component is primarily used to determine the current rover position. However, the Localization component which builds the occupancy map of the area also performs the current position estimate. Furthermore, since we deal with a fleet of rovers that can detect each other via their Localization components, we can also use that data to check the current position of each rover. To mitigate the risk of collision, we apply a mitigation control to cross check the position value across different inputs and either adjust the current position to handle any noise or make a full stop until we get a trustworthy position reading. Each mitigation control is associated with a number of requirements that detail how the control should be implemented. For example, the control in Figure 2 is related to the following requirement: If there is a large difference in position values between DWM, localization (SLAM) and Occupancy map positions, then the rover shall stop within 3 seconds.

We define the safety performance metrics based on the different events in the safety architecture as well as requirements attached to the mitigation controls. Examples of safety performance metrics include:

- Number of times self position error has been detected
- Number of times self position error has been detected and adjusted
- Number of times the rover has stopped because of the incorrect current position
- The trend of self position error in the last x seconds

Each safety performance measure can be viewed in the risk monitoring dashboard. However, we can also embed those in the safety architecture and structured arguments so that we can highlight parts of the safety architecture and arguments that require operators attention. For example, if an event in the safety architecture is happening much more frequently compared to the estimate used for risk analysis, we can add a condition and highlight that event. Furthermore, each mitigation control has an integrity at which it mitigates the hazard to which it is applied. The safety performance metrics can be used to highlight if the actual integrity of a control is different compared to the value used in the risk assessment.

B. Requirement Formalization

We specify requirements using FRET (Formal Requirements Elicitation Tool [7]), developed at NASA Ames Research Center. FRET is an open-source tool [14] for writing, understanding, formalizing, and analyzing requirements.

Create Requirement

Requirement ID	Parent Requirement ID	Project
DWM101-001		Troupe

Rationale and Comments

Rationale

If the difference in position values is greater than a certain threshold then the rover shall stop within a time bound of 3 seconds.

Comments

Requirement Description

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "**". For information on a field format, click on its corresponding bubble.

SCOPE	CONDITIONS	COMPONENT*	SHALL*	TIMING	RESPONSES*
If $\text{abs}(\text{dwm_position} - \text{slam_position}) > \text{threshold}$ DM shall within 3 seconds satisfy stop					

SEMANTICS

ASSISTANT

TEMPLATES

GLOSSARY

ENFORCED: in the interval defined by the entire execution. TRIGGER: first point in the interval if $(\text{abs}(\text{dwm_position} - \text{slam_position}) > \text{threshold})$ is true and any point in the interval where $(\text{abs}(\text{dwm_position} - \text{slam_position}) > \text{threshold})$ becomes true (from false). REQUIRES: for every trigger, RES must hold at some point with distance ≤ 3 from the trigger (i.e., at trigger, trigger+1, ..., or trigger+3). If the interval ends sooner than trigger+3, then RES need not hold.

Beginning of Time **TC**

$\text{TC} = (\text{abs}(\text{dwm_position} - \text{slam_position}) > \text{threshold}), n = 3,$
Response = (stop).

Diagram Semantics

Formalizations

Future Time LTL

Past Time LTL

Lustre

Format

```
H((OT(3, 3, ((abs((dwm_position - slam_position)) > threshold) and ZtoPre(not ((abs((dwm_position - slam_position)) > threshold)))) and not (stop))) => OT(2, 0, (not (YtoPre(true)) or stop))))
```

Fig. 3 FRET Create Requirement pane.

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 natural language named FRETish [15] with precise unambiguous meaning.

Let us look at a specific requirement from the TROUPE case study and how it can be specified and formalized with FRET. Figure 3 shows the ‘Create Requirement’ pane. The pane is split into two parts - the editor on the left and the assistant tab on the right.

We start by defining a unique name for the requirement in the ‘Requirement ID’ field of the editor. Requirements in FRET are organized in projects (see ‘Project’ field). Free text can be optionally entered in ‘Rationale and Comments’ that provide additional explanations of the specified requirement.

‘Requirement Description’ is the requirement written in FRETish. Each FRETish requirement is composed using up to six distinct fields (the * symbol designates mandatory fields): 1) **scope** specifies the time intervals where the requirement is enforced, 2) **condition** is a Boolean expression that triggers the **response** to occur at the time the expression’s value becomes true, or is true at the beginning of the scope interval (if $\text{abs}(\text{dwm_position} - \text{slam_position}) > \text{threshold}$, where $\text{abs}()$ is a function that returns the absolute value of a given argument), 3) **component*** (DM) 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 3 seconds) specifies when the response shall happen, subject to the constraints defined in **scope** and **condition** and 6) **response*** (satisfy stop) 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 and then FRET will generate English and diagrammatic explanations in the ‘Assistant’ tab. Additionally, FRET automatically produces formalizations in Future Time LTL and Past Time LTL. Through its analysis portal, FRET connects to analysis tools by facilitating the mapping between requirements and models/code, and by generating verification code. The connection

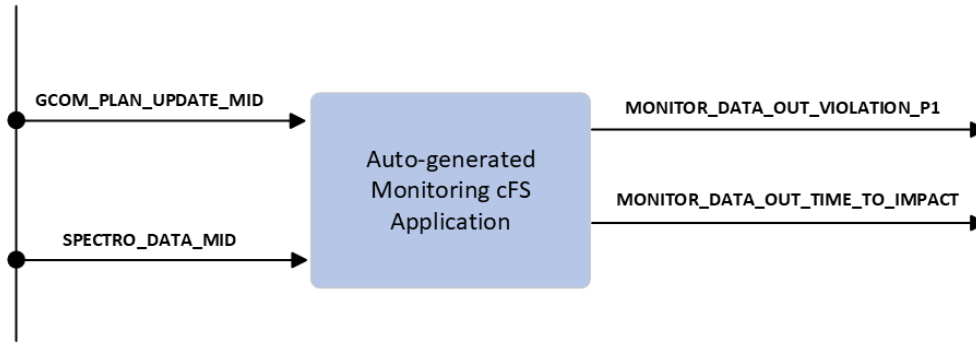


Fig. 4 High-level view of cFS monitoring application generated by Ogma.

that we use in this work is the one between FRET and the Ogma/Copilot tools [16, 17].

C. Generation of safety performance metric monitors as cFS apps

Ogma [8, 9] is a monitoring application generator that, among other features, is capable of generating cFS applications that gather information from multiple 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 the ground station (Fig. 4).

Copilot [10] is a runtime monitoring language that can be used to describe information and properties to be monitored using a high-level language. From those specifications, the Copilot compiler produces code that implements the monitors. Copilot can produce hard real-time C99 code or FPGA specifications, making it suitable to be used on embedded systems with limited resources.

From the assurance case, we can determine what information needs to be monitored, create the specifications needed, and use Ogma and Copilot 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 JKind Lustre [18] and FRET [14]. The second is a plain-text database that maps data by name to the cFS that produces such data. Such a database only needs to be compiled 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 dropped together with other cFS applications and require no modifications after being generated. In our architecture, such results are then obtained by the Telemetry Output (TO) application, which can relay them to the Ground Station for further processing.

III. Planned content

We plan to further detail the Dynamic safety case methodology. In particular, we will expand on the safety performance metric definition and its interface with monitor generation and the ground station software relying the information back to the assurance case. We plan to include the complete application example within Troupe, including the definition of the system and the safety architecture, the concrete performance metrics, requirements, the generated monitors and the results of the monitoring in the assurance case based on live demonstration.

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] Bourbough, 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] Ogma, <https://github.com/nasa/ogma>, Accessed: 2023-06-01.
- [9] 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.
- [10] Perez, I., Dedden, F., and Goodloe, A., “Copilot 3,” Tech. Rep. NASA/TM-2020-220587, NASA Langley Research Center, April 2020.
- [11] “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>.
- [12] Authority, C. A., “Bowtie risk assessment models,” , 2019.
- [13] 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.
- [14] FRET, <https://github.com/NASA-SW-VnV/fret/>, Accessed: 2023-06-01.
- [15] 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>.
- [16] 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.
- [17] 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.
- [18] 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.