An Engineering Life-Cycle Assurance Process for Autonomous Space Systems

Alessandro Pinto
NASA Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
apinto@jpl.nasa.gov

Caleb Wagner
NASA Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
caleb.t.wagner@jpl.nasa.gov

Klaus Havelund
NASA Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
klaus.havelund@jpl.nasa.gov

Nicolas Rouquette
NASA Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
nicolas.f.rouquette@jpl.nasa.gov

***Abstract*—**The increasing complexity of autonomous space systems, coupled with the desire to grow the space economy, necessitates standardized development practices that guarantee an assurance level commensurate with the high stakes of space missions, including cost, schedule, and potential loss of human life. Several standards have been published over the past decade, with some focusing specifically on autonomous driving systems on Earth. These standards advocate for an early analysis phase of the design specification, but they do not provide sufficient insights on how to define, decompose, and trace requirements – which is key in high-assurance systems. The autonomous driving industry relies on a large set of simulation scenarios and actual miles driven as evidence in their assurance argument. However, space missions must obey stricter safety requirements, and operational data might not be available, which suggests the need for a model-based, requirement-driven, and formal approach to design, test, and evaluation. This paper presents an engineering life-cycle approach to designing autonomous systems, and expands on two key elements: (1) the definition and decomposition of requirements and (2) the verification and validation of requirements through automatic test generation. We introduce a standard decomposition of an autonomous system into levels and layers, and we model its components using system- and control-theoretic methods. We propose a practical way to define executable specifications that can generate behaviors. Then, we present an automated test generation framework that comprises a monitoring system, several search and optimization algorithms, and a comprehensive data analysis framework. Finally, we show how the approach has been applied to a prototypical lunar rover mission.

TABLE OF CONTENTS

979-8-3315-7360-7/26/$31.00 ©2026 IEEE

1. INTRODUCTION

The design of new space exploration systems follows a rigorous process [1, 2, 3, 4] to minimize the risk of missing scientific goals, losing space assets, or even human lives. Many of these systems feature some level of autonomy as well, such as the Mars 2020 Perseverance rover. These systems have undergone a considerable verification and validation (V&V) effort [5, 6, 7]. However, risk reduction in these systems is primarily achieved with careful engineering of the hardware, the on-board fault protection system, and the V&V of command sequences on the ground. This approach has enabled the infusion of limited autonomous capabilities in space at a reasonable, albeit still considerable, V&V cost.

As the space industry ventures into more complex missions such as establishing a presence on the Moon or Mars, or exploring planetary bodies too far from Earth to allow for remote assistance, it is predicted that space systems will require a higher level of autonomy [8]. It is, therefore, important to understand whether current practices in systems engineering can still deliver the same level of assurance that has allowed the reliable operations of current systems.

There are several definitions of autonomy. NASA defines autonomy as "*the ability of a system to achieve goals while operating independently of external control*". This definition emphasizes the goal-oriented nature of an autonomous system and its independence from any external controlling entity. NATO provides a more detailed definition: "*a system that decides and acts to accomplish desired goals, within defined parameters, based on acquired knowledge and an evolving situational awareness, following an optimal but potentially unpredictable course of action*" [9]. This definition also refers to an independent process to make decisions and act. It also explicitly refers to the processes of acquiring knowledge and evolving situational awareness. Finally, the course of action, or plan followed by an autonomous system is optimal (or perhaps more generally "rational" [10]) and potentially unpredictable — with implied challenges for assurance. Essentially, an autonomous system is a closed loop system where the state evolves according to certain dynamics. One part of the system, which we call the System Under Design (SUD), is what is commonly (but imprecisely) referred to as the autonomous system, while the other part is the environment. Properties such as "reaching goal states", "guaranteeing safety" or "achieving some level of performance", are all stated over all possible (and unpredictable according to the

NATO definition) behaviors (i.e., sequences of states) of the system. Thus, an autonomous system is in fact just a system where the SUD has to be designed to meet certain properties when connected in a closed loop with the environment.

Despite the fact that autonomous systems are just systems, the few successful deployments in safety- or mission-critical applications are the result of design paradigms that effectively reduce the complexity and uncertainty associated with acquiring knowledge and making rational decisions. Automated warehouses and assembly lines rely on a well-characterized environment that has been conditioned to facilitate the reliable operation of robots in these applications. For example, visual markers in certain locations can be effective at managing localization uncertainty, while repetitive tasks based on predictable movements of high actuation authority robots minimize the probability of reaching undesirable states as the result of an action. Even in these cases, however, there is still room for accidents that may result in harm to people. This risk is mitigated by restricting access to certain areas, such as around a robot on an assembly line. With all these limitations, the autonomous system verification problem reduces to a control problem with limited uncertainty such as sensor noise. These kinds of problems can be solved precisely [11], and safety and performance bounds can be shown to hold via mathematical proofs, which is the strongest form of evidence in an assurance case (assuming that all models have been validated).

The design and analysis of these closed loop systems, however, can become complex due to several factors. The number of variables that characterize the relevant state of the environment and the SUD can become very large. Not all relevant variables can be observed directly and must be inferred from other observable variables, leading to epistemic uncertainty about the environment [12]. The dynamics of the closed loop system are not fully known at design time, are highly nonlinear, or a closed form solution for the system trajectories does not exist. This is certainly the case for most interesting terrestrial and space systems. For example, self-driving cars negotiate an environment with many other vehicles (human-driven and autonomous) and pedestrians whose intents are not observable. The SUD includes several subsystems and must respond to a large variety of situations, from slippery roads (that can be dealt with by using advanced control systems) to flat tires, mechanical failures, burned out light bulbs, or other electrical failures. These discrete events require changing the driving mode and making many other considerations that render the dynamics of the whole system difficult to characterize (also because any change in the driving mode, such as a minimum-risk-maneuver to pull to the right, triggers changes in how the environment behaves in response). After more than 15 years of development, Waymo has been able to deploy autonomous cars in pre-mapped environments, with fleet response operators ready to intervene. However, changing driving regime such as traveling on a freeway, or in a different city, requires a considerable investment in pre-characterizing the environment, and testing for millions of miles. Neither is possible for space systems.

Space systems, and in particular exploration systems designed for scientific missions in less-known environments, need to manage energy, thermal states, communications, motion, instruments, and sample collection. Many of these sub-systems are critical to the success of a mission. Their actions must be coordinated, and resources such as energy must be managed to avoid preventing critical actions from being executed at the right time. These systems must be able to detect and react appropriately to a large number of off-nominal situations. However, there are important differences between self-driving vehicles and space exploration robots. The first is the level of criticality of the system. Both operate as safety-critical systems, but their stakes differ. For self-driving cars, the overriding concern is human safety, where any loss of life or serious injury is unacceptable. For space rovers, the primary concern is mission success and asset preservation, where a single collision or hardware failure on another planet could mean the end of a multi-billion-dollar mission with no possibility of repair. The second important difference is in the evidence to support an assurance case. The self-driving industry can rely on data collected in the operational environment, accurate simulations, or historical accident databases to provide statistical metrics of performance and reliability. The design of a space system must rely more on models, also because of lack of access to the operational environment.

There have been several important efforts to provide guidance for the design and verification of autonomous systems. The most relevant to the work presented in this paper are the ISO 21448 "Safety of the intended functionality (SOTIF)" [13], and the System Theoretic Process Analysis (STPA)[14]. The SOTIF standard, which has been developed for the automotive industry, provides a methodology to assess the intended functionality of a system with respect to the potential harm caused by functional insufficiencies rather than functional failures (which is covered by ISO 26262 [15]). The methodology is divided into two steps: identification and mitigation of known hazards via analysis, and identification and mitigation of unknown hazards via stress testing. The first step requires the development of models for the operational design domain, the systems, and the causality chain from a functional insufficiency (e.g., vision system performance when facing the Sun) to harm (e.g., loss of life).

STPA proposes a model where the behavior of a system of interacting sub-systems is controlled by a higher level system which is decomposed into a process model that observes the system and estimates its state, and a controller that sends actions to the controlled system. The methodology provides a structured way of identifying unsafe control actions and tracing them back to their causes. STPA can be applied to a variety of systems, and it is an analytic method used at design time, or as a way to identify the root cause of an accident.

In this paper, we take inspiration from these previous works, and we make the following contributions:

- We formalize a life-cycle approach to assurance of autonomous systems that is particularly suitable for space mission design and V&V. While we recognize the importance of data, we also emphasize the need for creating models since data may be scarce or not available for new missions. We also cannot rely only on testing due to the inaccessible operational environment.
- We define a modeling approach for autonomous systems that drives the definition and traceability of requirements, and defines the V&V activities at different levels of abstraction of the system. This model is inspired by the STPA approach, but it is formal and amenable to automation.
- We provide practical implementations of tools to address two important steps in the methodology:
  - *Requirement generation and validation*. We assume that the goals of the space mission have been defined and that the system has been decomposed into its first level components. We are not concerned with the design of the hardware

such as the thermal system, the propulsion system, or the power system. Rather, we are concerned with the control software component. We assume that high-level capabilities have been assigned to the software. This step can leverage methodologies such as goal-based requirements engineering [16]. From this specification, we provide a methodology to create a high-level model of the intended functionality of the system which is the composition of control systems, each in charge of controlling a sub-system on the spacecraft. Then, we show how the system can be analyzed either by simulation or by formal methods. This first step yields a set of requirements that the situational assessment and decision-making algorithms must satisfy.

– *Automatic test generation*. The second key technology that we use is test generation. We assume that the system has been implemented and that we have access to a simulator of the environment. We then use test generation to find scenarios in which the system does not satisfy the requirements identified in the first step.

• We apply the methodology to a lunar rover mission that comprises a system-level autonomy layer and three subsystems abstracted by their intended functionality: the navigation system, the communication system, and an instrument to perform scientific measurements. We discuss results and lessons learned.

2. LEVELS AND LAYERS IN AUTONOMOUS SYSTEMS ARCHITECTURES

Control hierarchies arise in nature, society, and technical systems. Hierarchies have also been proposed by most of the architectural frameworks for the design of autonomous systems [17, 18, 19]. More recently, various instantiations of layered control architectures have also been formally studied [20, 21, 22, 23]. Hierarchical control systems become necessary to deal with the complexity of making decisions, but they are also necessary for modularity and for reusing well-characterized behaviors. Without an organized structure, each design would need to consider how decisions affect each other and resolve them by negotiating among several decision makers. For example, the navigation subsystem of a rover would need to know when the next communication window will start and decide whether to continue driving or stop. A higher level of control, instead, could reuse these independent capabilities to generate many possible sequences of driving, stopping, and communication tasks.

At the lowest layer of the hierarchy, we find the control system that is responsible for regulating physical variables in the environment. This layer is implemented as a set of feedback control loops that ensure that the system tracks a reference trajectory while satisfying constraints on the value of the state and input variables. The layers above the control layer are often application-dependent. In [20], for example, the authors develop models for trajectory planning (optimization-based), and decision-making (logic-based). In [24], the authors report the hierarchical architecture of the Endurance [25] lunar rover with particular emphasis on the decision-making layer which is partitioned into a ground layer and an onboard system-level deliberative layer. The latter interfaces with a number of lower-level layers, one for each subsystem, including mobility and communication.

Layers shall be distinguished from levels [26]. While we identify layers as service providers to the layers above (or users of services provided by the layers below), levels refer to the granularity of the representation of an architecture. It is essential to leverage levels as much as possible to reduce the complexity of the design and verification activities. The definition of the layers and levels of an architecture evolves together with the definition of the requirements for each component in the system and establishes traceability links. This is an essential part of our methodology.

Figure 1 shows a high-level view of the levels and the layers that our methodology leverages in the design and analysis of an autonomous system. The first level, lv_0, is our starting view of the system as the feedback interconnection of an environment and the SUD. The system behavior depends on a set of parameters P (such as the maximum speed and power draw of a rover on the Moon), and can be observed through a set of observable outputs O (such as the state of charge or the current task being executed). The environment is always considered the lowest layer ly_0. We cross levels by *refining* the previous layer. Level lv_1 shows a refined view of the SUD, which is decomposed into two layers: the ground controller and the rover system. This refinement is denoted by R_0 in Figure 1.

Note: Refinement links. Refinement links are traceability links that can be formally defined using a compositional framework as shown in [22]. Consider R_0 as an example. We can characterize the environment with an assumption $A_{E,0}$ representing the set of command sequences that can be accepted from the SUD, and a guarantee $G_{E,0}$ representing the way in which the environment responds to commands and provides observations when the assumption $A_{E,0}$ is met. For example, $A_{E,0}$ may impose bounds on the maximum commanded accelerations, while $G_{E,0}$ specifies bounds on the error of the reported position of the rover. When all components are specified this way, contract-based design [27, 28] can be used to formally check the compatibility of components and their consistency and completeness across levels. Intuitively, the SUD will have its own assumptions $A_{SUD,0}$, which are often called environmental assumptions, and which must be met by $G_{E,0}$[1]. Refinement R_0 must maintain assumptions and guarantees in the following sense. The environment at level lv_1 must have a larger assumption set, i.e., $A_{E,0} \subseteq A_{E,1}$, and must have an equal or smaller set of guaranteed behaviors $G_{E,1} \subseteq G_{E,0}$ (please refer to [28] for a precise definition of the refinement relation). All refinement relations introduced later can be formally verified using this approach. □

Finally, the composition of the ground controller and the rover system must be a refinement of the SUD. The next refinement R_1 decomposes the rover into a system-level control layer and a set of dedicated controllers for different subsystems such as navigation, communication, and instrument control.

Each of these sub-systems can be further refined into a (not necessarily uniform) number of functional layers (e.g., from ly_1 to ly_f as shown in the figure). For example, the navigation layer can be decomposed into a path planner, a trajectory tracking layer, and a motor control layer. Thus, in reality, *the layered architecture of a system is a tree where each path is a sequence of layers*.

Note that we have used an implicit definition of a layer. In our modeling approach, a layer is really a component

[1]Notice that in a feedback composition, we must avoid circular reasoning. In this case, for example, $G_{E,0}$ is guaranteed by the environment only if $A_{E,0}$ is satisfied. But $A_{E,0}$ is satisfied only if $G_{SUD,0}$ is satisfied, which in turn depends on the satisfaction of $A_{SUD,0}$.

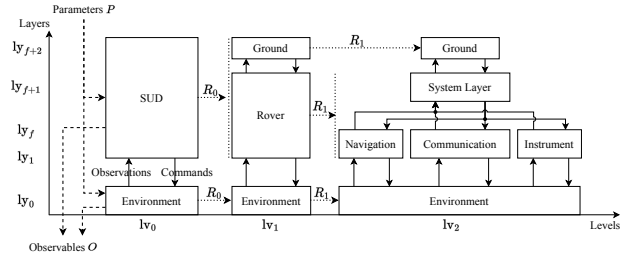

Figure 1: The definition of the layers and levels that we consider in our methodology.

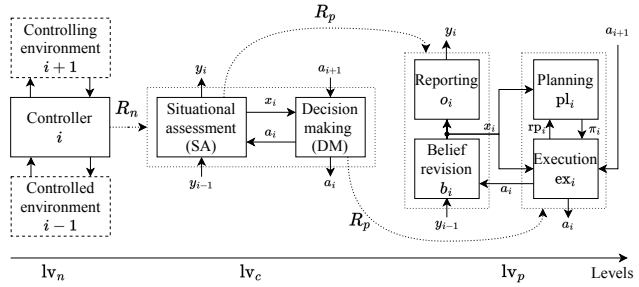

Figure 2: Refinement of a controller into a the control level lv_c, and the further refinement of the control level into the policy level lv_p.

that "refines" behaviors. The command requests it receives from the layer above are refined into sequences of command requests for the layers[2] below. This is the typical planning problem [29] where a command can be identified by a goal to be reached (i.e., bringing the value of the observables at some point in the future within a given goal set g), or by a task to be executed as specified in hierarchical task networks [30]. The decomposition of a layer into sub-layers continues until the planning problem has a practical solution. As an example, if we stopped our decomposition at level lv_1, we would need to solve a planning problem to refine a task such as "reach science sites $\text{PoI}_1, ..., \text{PoI}_n$ in sequence while communicating with the ground" into motor control signals. First, such a problem cannot be solved monolithically in a practical way. Second, the solution would be too rigid to also address other potential commands from ground such as "driving to location PoI_j while taking measurements every d kilometers".

When a layer reaches a level where an implementation can be attempted, then it can be further refined into lower levels of abstraction as shown in Figure 2. Consider a controller i. It has two environments: the controlling environment, which sends commands a_{i+1} based on the observations y_i received from i, and a controlled environment that generates observations y_{i-1} based on the commands a_i sent by controller i. The next level of abstraction, here denoted lv_c, is a classical decomposition [31, 32] of controller i into two components: situational assessment (SA) [33] and decision making (DM). The responsibility of SA is to maintain an internal estimate x_i of the state of the environment based on the observations y_{i-1} and the actions a_i. The responsibility of DM is to select the next action a_i based on the command a_{i+1} and the current state estimate x_i.

[2]Notice that we used the plural "layers" here because a layer may send commands to multiple other layers, each controlling a different sub-system.

Level	Requirements
lv_0 - lv_1	Up to subsystem requirements (L4)
lv_1	Autonomy capabilities (LA)
lv_2 - lv_n	Controller interface requirements (LCO)
lv_c	Belief and Policy requirements (LBP)
lv_p	Algorithmic requirements (LALG)

Table 1: Mapping between the levels and the set of requirements defined at that level.

We call the next level lv_p, where SA and DM become more explicit and algorithmic design decisions must be made. SA is decomposed into a belief revision component b_i and an observation generation component o_i. Finally, DM is decomposed into a planning component pl_i and an execution component ex_i. Upon receiving a plan generation command rp_i that carries the current command from the controlling environment to be pursued next, the planning component generates a plan π_i starting from the current state x_i. The plan is then given to the execution component which uses it to generate appropriate commands a_i as a function of the evolving state estimate x_i. We make two important observations. First, the command carried by rp_i does not have to be the most recent command from the controlling environment. It is up to the execution algorithm to decide the high-level command to be pursued next. Second, the execution component may decide to generate a new plan even if there is no change in the command from the controlling environment. This may happen when the state estimate is in conflict with the prediction that was made at planning time.

3. OVERVIEW OF THE METHODOLOGY

The goal of the methodology is to define a validated requirement tree that satisfies the goals of a space mission. Table 1 shows the different sets of requirements that we define at each of the levels described in Section 2 (Levels and Layers in Autonomous Systems Architectures).

lv_0 is already part of a standard development process. In fact, it corresponds to the definition of system and subsystem-level requirements. The component that controls the system (the SUD) is a single entity that observes the state of the environment (which includes the system's hardware components), makes decisions, and sends commands to the actuators. The requirements allocated to the SUD tend to be expressed in the form of high-level capabilities such as "take a high-quality measurement of property X at location Y". The definition of these requirements is a collaborative effort among many stakeholders including scientists and subject-matter experts (SME) from a wide variety of engineering disciplines. The specification language used in this phase tends to be English, as it is universally understood by such a heterogeneous team.

The capability requirements derived at lv_0 are partitioned at lv_1. The engineering teams allocate roles and responsibilities to the ground and onboard systems and define the interface between them. An initial concept of operations (ConOps) is taken as input and refined at this level. The result of this step is the definition of the commands a_{f+2} from the ground system to the onboard systems and the observations y_{f+1} in the opposite direction[3].

[3]In this particular example, f is the number of layers between the environment and the top-layer of the on-board system hierarchy (excluded).

The decomposition at the next level, called lv_2, results in the definition of the control layers for the entire system and the specification of the commands and observations for all components. The starting point for the definition of this tree of control systems is the set of constraints derived at lv_1, which includes (1) the expected behavior of the system when executing command a_{f+2} and (2) the assumptions that the environment must satisfy. Our methodology provides several tools to define these interfaces that impose behavioral requirements for each controller. We start by defining commands.

A command specification comprises a name a, a set of parameters $\theta(a)$, a set of fluents $V(a)$ (time-varying properties) also called state variables in the scope of the command, a dynamic model $\text{dyn}(a)$, a goal criterion $g(a)$, an error criterion $e(a)$, and a set of metrics $m(a)$. The dynamic model defines the set of possible behaviors for the command, where a behavior is a particular evolution of the values of the state variables $V(a)$ over time, starting from an initial value taken from a set of possible initial conditions $V_0 \in pre(a)$. Here, $pre(a)$ is a predicate over $V(a)$ called the precondition. We write $V(a,t)$ to denote the value of the state variables under the scope of a at time t. The end criterion is a predicate over $V(a)$ that determines when the command has ended. This means that we don't consider commands as events, but rather as durative actions [34] (albeit the duration may or may not be explicitly defined). The execution of a command may stop when the error or goal criteria are met, but this is up to the controller that is executing the command and is part of the decision making policy. Finally, $m(a)$ is a function that maps partial behaviors over $V(a)$ to quantities of interest such as energy usage. The metrics are used to rank the possible behaviors in order of preference. The elements of a command $\text{dyn}(a)$, $g(a)$, $e(a)$, $m(a)$ are all parametric in $\theta(a)$. Each choice of values for the parameters results in a different instance of the command.

Consider a command message a_{i+1} produced by a component c_{i+1} at layer $i+1$ and received by a component c_i at layer i. These two components reason over different state spaces and may use different models. Thus, we need to distinguish two views of a command a_{i+1}. The *controlling environment view* $a^{\top}_{i+1}$ represents the assumption that c_{i+1} makes about the impact of the command message a_{i+1} on the variables $V(a^{\top}_{i+1}) \subseteq x_{i+1}$. The *controlled environment view* $a^{\perp}_{i+1}$ represents the guarantee that c_i provides on the change of $V(a^{\perp}_{i+1}) \subseteq x_i$ when a command message is received. Clearly, the guarantee must support the assumption, which can be checked after eliciting the command models and assessing their compatibility (or evaluating the potential effect of their incompatibility).

To facilitate the complete definition of these command views, we leverage a list of analysis templates that elicit the definition of nominal and off-nominal situations. Component c_i will be designed to generate commands for its controlled environment in order to satisfy the command constraints imposed by $\text{dyn}(a_{i+1})$. The process that transforms a_{i+1} from the controlling environment of c_i to commands a_i for its controlled environment is the decision-making process shown in Figure 2. When building a command model for $a^{\perp}_{i+1}$, we perform the following analyses:

- *Planning time failure analysis*:
  - *No progress analysis*. This analysis identifies values of the state variables $V(a^{\perp}_{i+1})$ in which c_i rejects command a_{i+1} without attempting to generate any plan. For example, a command to drive is rejected if the state x_i indicates that the rover position is too close to an obstacle to attempt any motion. This analysis contributes to the definition of $pre(a^{\perp}_{i+1})$.
  - *Partial progress analysis*. This analysis identifies values of the state variables $V(a^{\perp}_{i+1})$ in which some progress towards the goal $g(a^{\perp}_{i+1})$ can be made, but the goal cannot be reached. For example, a goal location is enclosed by obstacles, or the designer of c_i knows that the decision making problem is too hard to be solved exactly, and that any reasonable heuristic will not be complete (i.e., the planning algorithm may not be able to find a plan, even if one exists). This analysis contributes to the definition of $e(a^{\top}_{i+1})$.
- *Execution time failure analysis*:
  - *No progress analysis*. This analysis identifies situations during the execution of the command $a^{\perp}_{i+1}$ that may reach a state from where the system can no longer make progress. These are situations that cannot be predicted at planning time due to insufficiencies in the models used for planning or in the perception system. This analysis contributes to the definition of $e(a^{\top}_{i+1})$.
  - *Discrepancy between x_i and the desired behavior defined by dyn*$(a^{\perp}_{i+1})$. This analysis identifies cases in which the state estimate x_i does not track the behavioral constraints imposed by $\text{dyn}(a^{\perp}_{i+1})$. This analysis contributes to the definition of $e(a^{\top}_{i+1})$.
- *Prediction failure analysis*:
  - *False negative analysis*. This analysis aims at defining an appropriate model $\text{dyn}(a^{\perp}_{i+1})$ to limit the misprediction of future execution time failures. The analysis identifies cases where the model predicts a failure in the future, but the failure would not have occurred had the system continued the execution of the plan.
  - *False positive analysis*. This analysis aims at defining a appropriate model $\text{dyn}(a^{\perp}_{i+1})$ to limit actual execution time failures. The analysis identifies cases where the model predicts the achievement of the goals in the future, but a failure will occur if the system continues the execution of the plan.

Each layer is responsible for monitoring the execution of the commands that it issues to the controlled environment. This is essential to advance the execution of a plan, to recover from failures, and to change plans when progress is not satisfactory. This means that each layer must be able to identify when a command succeeds or ends with an error, and whether the command is satisfying its model. For example, c_{i+1} must be able to assess whether $e(a^{\top}_{i+1})$ is true or false. More generally, the specification of a command a_{i+1} results in a set $Prop(a_{i+1})$ of pairs of properties $(p_{i+1}(a^{\top}_{i+1}), p_i(a^{\perp}_{i+1}))$ where $p_{i+1}(a^{\top}_{i+1})$ is evaluated over $V(a^{\top}_{i+1})$, and $p_i(a^{\perp}_{i+1})$ is evaluated over $V(a^{\perp}_{i+1})$. The controller issuing the command (in this case c_{i+1}) must be able to assess whether $p_i(a^{\perp}_{i+1})$ is true. This important requirement drives the definition of the observations y_i at each layer of the hierarchy. The following analysis tasks provide a way to arrive at their preliminary definition:

- *Epistemic uncertainty analysis*. This analysis aims at determining whether knowing that $p_{i+1}(a^{\top}_{i+1})$ holds is sufficient to establish that $p_i(a^{\perp}_{i+1})$ also holds. To perform this analysis, the relation between x_i and x_{i+1} must first be defined. Given a state x_i, the analysis consists in deciding

how the state is measured and reported through observation y_i, and how the observation may affect x_{i+1}. The inverse relation usually identifies a set $X^{-1}(x_{i+1})$ that represents the possible values of the state x_i for a given value of x_{i+1}. Given that $p_{i+1}(a_{i+1}^{\top})$ holds, then there is epistemic uncertainty if $p_i(a_{i+1}^{\perp})$ holds for some states in $X^{-1}(x_{i+1})$, but not for others. Epistemic uncertainty can arise from partial observability of the controlled environment, or insufficiencies in the ability of a controller to deduce the state from the observations.

- *Aleatoric uncertainty analysis*. This analysis assesses the statistics associated with a property of the state. The analysis determines the expected values and variances of the two properties $(p_{i+1}(a_{i+1}^{\top}), p_i(a_{i+1}^{\perp})$ to determine if the state is more or less uncertain at the higher level of the hierarchy.
- *Delay-induced uncertainty*. There are always delays involved in the loop from a command to the observation of its effect. This analysis determines the expected delay δ and establishes whether $p_{i+1}(a_{i+1}^{\top}, t)$ can still be used to infer $p_i(a_{i+1}^{\perp}, t + \delta)$. In some cases, the behavior of c_{i+1} is not affected by the delay, or a prediction can be made to mitigate its effect.

An additional analysis is performed to define any additional constraints on sequences of commands rather than on a single one. This analysis is meant to assess whether controller c_i imposes any of the following typical constraints on the command sequence sent by c_{i+1}:

- c_{i+1} can/cannot send the same command before c_i has completed the execution of the previous one, depending on the resulting state (e.g., whether the state indicates a successful execution or an error). For example, a command to drive to a specific location cannot be followed by the very same command while the rover is driving. If c_i does not impose this constraint, then the resulting behavior (e.g., overriding) must be defined.
- A command a_{i+1} must always be preceded/followed by a command a'_{i+1}. For example, a drive command must always be preceded by a pre-heating command to warm up the actuators involved in driving.
- A command a_{i+1} shall not/shall overlap with another command a'_{i+1}. For example, a communication command shall not overlap with a driving command.

These constraints often require the definition of additional states in the system to keep track of the history of commands sent to the controlled environment.

After all interfaces between controllers and behavioral constraints have been defined, a V&V activity is required to check that the controlling environment and controlled environment views of a command are compatible, and that each controller is able to monitor the execution of its commands. Furthermore, an argument must be made that the capabilities defined at lv_1 can be delivered by the system (i.e., refinement relation R_1 holds for the rover system).

The next step is to define constraints at level lv_c to restrict the behaviors of a controller and to satisfy the constraints derived at lv_n. By behavior, we mean the sequence of commands sent to the controlled environment in response to the commands sent by the controlling environment, and, as a result, the evolution of the state variables of the SUD and the environment as a function of time. At this level, additional constraints may be generated to manage shared resources such as energy.

Note: Reference model at level lv_c The standard model at level lv_c can be formally defined[4]. The SA model is a relation between its inputs (y_{i-1}, a_i, x_i) at time t and its outputs (y_i, x_i) at the next time step. This relation can be captured by a function:

$$\text{SA}_i : \text{Dom}(y_{i-1}) \times \text{Dom}(a_i) \times \text{Dom}(x_i) \rightarrow 2^{\text{Dom}(x_i) \times \text{Dom}(y_i)}$$

Where $\text{Dom}(x)$ denotes the domain of values of variable x. As mentioned in Section 2, commands are processes that span a certain amount of time. Thus, one of the state variables that we always consider is the current set of commands that are "active", meaning they have been sent to the controlled environment and have not met an end criterion yet.

The decision making function is a classical permissive policy [35]:

$$\text{DM}_i : \text{Dom}(x_i) \times \text{Dom}(a_{i+1}) \rightarrow 2^{\text{Dom}(a_i)}$$

While a specific plan leads to a deterministic policy, level lv_c is not low enough to select a specific course of action for a command. The constraints on DM_i should actually be as loose as possible and only represent the essential constraints to ensure that key requirements are satisfied. This is why we rely on permissive policies. These policies can be specified directly by the function DM_i, or they can be specified as behavioral constraints from which a maximally permissive policy can be automatically synthesized [35, 36]. In Section 5 we present a tool that has been developed to allow for the simulation of the system at level lv_c. □

The last level before software implementation is lv_p. At this level, engineers decide how to best represent commands and constraints, how to implement the decision-making constraints in the form of planning and execution algorithms, how to design a belief revision system that satisfies the epistemic and aleatoric uncertainty constraints, and finally how to ensure that the delay bounds are satisfied. A key verification problem is to check that the composition of pl_i and ex_i satisfies the decision making constraints derived at lv_c.

4. LUNAR ROVER EXAMPLE

We introduce an example inspired by the Endurance mission [25]. In this mission, a rover must collect samples from various regions in the Moon's South Pole-Aitken basin. The sample collection operations are supervised remotely by the ground team. The mission goals and the allocation of roles between the ground and the rover system can also be found in [25]. We will focus on lv_2 and lv_c. Figure 1 shows a high-level diagram of the rover's layers at level lv_2. Command family $a_{f_{+2}}$ is a broad class (in practice captured by specific types of task networks [37]). We focus on a subset of this family named $drives$, with parameters $\theta(drives) = (wp_1, .., wp_k, \Delta_m, \Delta_d)$, where $(wp_1, .., wp_k)$ is a list of waypoints and Δ_m and Δ_d are two additional parameters explained below. The expected behavior of the rover system must satisfy the following constraints (which essentially define $\text{dyn}(drives^{\top})$):

- The rover must pursue the waypoints in the given sequence.

[4]In this short paragraph, we will use a non-deterministic formulation. A probabilistic formulation is possible by replacing the range of the SA_i and DM_i functions with probability distributions.

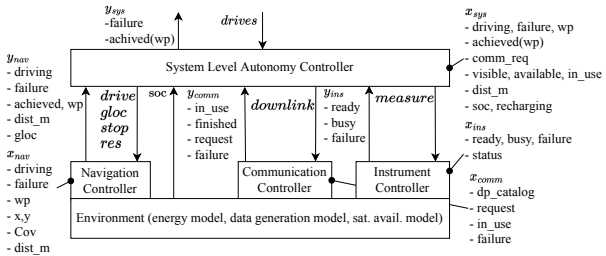

Figure 3: Details of the lv_2 model of the rover. The command, state, and observation.

- The rover must communicate telemetry and scientific data at the beginning of each availability window.
- The rover must take measurements periodically during its traverse every Δ_m kilometers (e.g, $\Delta_m \in [1.8, 2.2]$).
- The rover shall maintain a localization error no greater than Δ_d.

The goal criterion is that all waypoints are eventually reached and that communication windows and measurements are never skipped. The execution fails whenever the rover can no longer drive or take measurements. Communication is vital, and the rover should be designed so that the probability that the communication system fails is acceptable.

Figure 3 shows the result of the modeling effort according to our methodology. The navigation controller is in charge of moving the rover to a given waypoint. Thus, it must accept a $drive$ command with an identifier of the location to reach and a $stop$ command to stop driving. The drive model for $drive^{\perp}$ was elicited using the methodology described in Section 3, resulting in the identification of the following failures:

- Planning time failures: already driving, already at destination, actuator failure, no obstacle-free path to destination found, too close to an obstacle, already in a failure mode, not enough energy to drive.
- Execution time failures: failure to track the desired path, unexpected large covariance in the state estimate, actuator failure, too close to an obstacle, no obstacle-free path to destination exists, not enough energy to drive.

The precondition of the $drive^{\top}$ command can be developed based on these observations. For example, $\neg driving \wedge \neg failure$ must be part of the precondition. However, the system-level controller does not observe the map of the environment or the location of the rover, and cannot assess whether the rover is close to an obstacle. Thus, there is no way to define a precondition $pre(drive^{\top})$ that implies $pre(drive^{\perp})$. This epistemic uncertainty means that there could be a case in which a drive is rejected (for reasons other than the rover being already driving or in a failure state). Finally, the dynamic model dyn$(drive^{\top})$ simply states that the $driving$ state should be true while driving. However, after a delay analysis, it is easy to see that such a model would not be correct. A better model is $(t > t_s(drive) + \delta_{drive}) \Rightarrow driving$, meaning that if the current time is greater than the time when the command was sent, $t_s(drive)$, plus the loop delay, then the rover should be driving.

The navigation controller is the only component that maintains an estimate of the position of the rover. It must also maintain an estimate of the uncertainty (called covariance) around the position estimate. For this reason, the navigation system is also assigned the task of monitoring the growth of the uncertainty over time and sending an observation $gloc$ (a boolean value) which indicates the need to perform global localization. The system-level controller can send a global localization command $gloc$ to the navigation controller.

Given that the navigation controller maintains a position estimate, it is also assigned the responsibility of maintaining and reporting a state $dist_m$ that represents the distance traveled since the last measurement was taken . The navigation controller, however, does not directly know when a measurement has been taken. Thus, it must also support a command res to reset such a state.

The other two commands sent by the system-level controller are: $downlink$ (to the communication controller) and $measure$ (to the instrument). The communication controller periodically queries a data product catalog that maintains a list of data products with their associated data sizes to be downlinked to the ground. While some data products are always present (such as telemetry) and must always be communicated, our model also allows for an explicit observation $request$ to download extra data. The communication controller also reports when the data link is in use. Similarly to the drive command, a model for $downlink$ is simply that after a loop delay δ_{comm}, the in_use observation becomes true. The $downlink$ command can fail for several reasons, but clearly, it fails when the relay satellite is not available. For this reason, the system level state x_{sys} includes variables $visible$ and $available$ that are true when the relay satellite is visible and available for communication, respectively. The explicit addition of these states removes the possible uncertainty in evaluating the precondition of the downlink command.

The last analysis is concerned with eliciting constraints among command sequences and any additional constraints to manage resources such as energy. The list of constraints that we have elicited is as follows:

- *Command dependencies*:
  - A $drive$ command should not be issued if the rover is already driving.
  - A $downlink$ command should not be issued if the satellite is already in use.
  - A $measure$ command should not be issued if the instrument is in use.
  - When the rover is driving, a $stop$ command must be issued before a $gloc$, $downlink$, or $measure$ command.
  - $drive$ should never be issued while executing $gloc$, $downlink$, or $measure$.
  - When using a radiometric global localization method, $downlink$ and $gloc$ should not overlap, and $gloc$ should only be issued if the satellite is visible.
  - When using optical global localization methods, $gloc$ and $measure$ should not overlap.
  - $gloc$, $downlink$, and $measure$ should never be interrupted before ending.
- *Energy management constraints*:
  - $drive$ shall not be issued if the state of charge is below a minimum threshold: $soc \leq soc_{min}$ (where soc_{min} is a parameter).
  - The rover should be placed in sleep mode when $soc \leq soc_{low}$ (where soc_{low} is a parameter).
  - When in sleep mode, no command should be issued until $soc \geq soc_{high}$, except for the $downlink$ command.

The state x_{sys} has been defined to support the definition of

DM_{sys} in such a way that the constraints listed above can all be satisfied. We have developed a modeling library to facilitate the definition of these constraints, and to enable the generation of behaviors for a system specified up to level lv_c. We will describe the simulation environment in Section 5.

5. ANALYSIS AT THE LV_c LEVEL

The methodology described in Section 3 guides the developer in the definition of commands, observations, and lv_c constraints. This level of detail is sufficient to develop tools that explore the space of possible behaviors. This set of behaviors is larger than the set of behaviors that the system at level lv_p generates, since the functions DM and SA are underspecified. For example, we have not imposed any constraints on optimal choices. The refinement of permissive policies into deterministic policies is done by the planning system designed at level lv_p. However, critical properties, such as the ones defined in Section 4, can be verified for all behaviors at level lv_c, regardless of the implementation choices made at the subsequent levels.

We are pursuing two paths to explore the set of possible behaviors: simulation and formal methods. The simulation environment we have developed requires implementing executable specifications of the DM and SA components, which can be a difficult task depending on the number of constraints for each component. The advantage is that a simulation model can be written in an imperative programming language such as Python and is therefore widely accessible to engineers who are not formal methods experts.

Simulation Model

The simulation environment is written in the Python programming language. It provides several services to define controller components with their input and output ports and to connect these ports to form a system. Ports are buffered channels, and components execute according to their assigned rates.

Each time a controller c_i runs, it executes the following steps:

- Read all input observations.
- Use function b_i to generate a set of possible next states as a function of the input observations and the previous state. This function is user-defined and should be implemented based on the analysis done in Section 3.
- Select one next state and generate the corresponding observations using function o_i. This function is user-defined.
- Read all input commands.
- Execute the DM function, which is decomposed as follows:
  - Select a `Policy` depending on the input commands and current state. The policy selection function is user-defined.
  - Update the set of active commands (some of the active commands should be deactivated depending on the end criteria).
  - Apply the selected `Policy`, which returns a set of possible commands. The simulation environment provides base classes to define policies that encode the rules derived in the analysis step (see Section 4 for examples).
  - Select the commands to activate among the possible commands, update the list of active commands, and send the commands to the controlled environment.

We present some details on the policy definition support that the simulation environment provides to users. A `Policy` is defined by rules, where each rule is a function that takes as parameters a set of candidate commands `candidate:list[Command]`, the current state of the system `s:State`, and the set of active commands `active:list[Command]` and returns the subset of the candidate commands that satisfy a certain condition. For example, the following rule removes a $drive$ command from the list of candidates if the state of charge is below a minimum threshold.

```
def stop_if_no_energy(
    candidate:list[Command],
    state: SystemLevelState,
    active: list[Command]):
    return [x for x in candidate
            if not (isinstance(x,DriveCommand)
                    and state.soc <
                    state.conf["soc_min"])]
```

To define a policy, the user can compose these rules in a serial or parallel fashion. The serial composition of two rules results in the intersection of their outputs, while the parallel composition results in the union. A policy, then, is the serial composition of a list of parallel compositions of rules (where a parallel composition of a single rule is the rule itself).

The policy used by the system level controller described in Section 4 is shown in Figure 4. In particular, the user defines a list called `rules` (line 13). The elements of the list are considered in series, while each element is a parallel composition of rules. The `Policy` base class offers a single function, `apply`, that takes the list of candidate commands, the state, and the active commands, applies all the rules, and returns the candidate commands that have not been filtered out. The user is responsible for the definition of the candidate commands when applying a policy.

Line 13 adds a standard rule that removes commands if their precondition is not satisfied in the current state. Depending on the policy type, the user adds different types of rules. For example, in the case of radiometric global localization (line 21), the $gloc$ commands can be allowed only if the satellite is visible (line 22), if $gloc$ can finish within the visibility window (i.e., has the opportunity to complete successfully), and can finish before the next availability window (line 24) where $downlink$ must happen. Also notice on line 34 the parallel composition of three rules to stop the rover: a communication command must be issued, the energy level is too low, or either $gloc$ or $measure$ are supposed to occur.

We have developed a full model of the system shown in Figure 3, using abstract models for the dynamics, perception, and energy consumption of the rover. Figure 5 shows the typical waveform diagram that can be obtained after a simulation run. Each sub-plot represents the evolution of a state over time. The first subplot shows when the rover is driving. The second subplot is the position covariance cov (localization uncertainty) at the navigation level . The covariance triggers global localization tasks $gloc$ (3rd subplot). The 4th subplot is the $dist\_m$ value which triggers a measurement task $measure$ (5th subplot). The 7th subplot shows the communication task execution that uses the satellite and that occurs always within an availability window (6th subplot). Finally, the last two subplots represent the state of charge soc and the recharging task respectively.

Formal Methods

As discussed above, we model the autonomous system directly in Python because it is highly expressive, is executable, is supported by a rich collection of libraries, and is user-

```python
class SystemLevelPolicy(Policy):
    def __init__(self, t,
                 contact_graph : ContactGraphTimeline,
                 measurement_duration : float,
                 gloc_duration: float,
                 policy_type : str = "RADIO_GLOC_AND_EXCLUSIVE_COMM",
                 name="UNNAMED"):
        super().__init__(t, name)
        self.cg = contact_graph
        self.mesurement_duration = measurement_duration
        self.gloc_duration = gloc_duration
        self.policy_type = policy_type
        self.rules = [[command_precondition_rule],[communicate_only_to_completion(contact_graph)]]
        match policy_type:
            case "RADIO_GLOC_AND_EXCLUSIVE_COMM":
                self.rules.extend([[measurement_can_end_before_next_comm(contact_graph,
                                                                         measurement_duration)],
                                   [gloc_only_when_sat_is_visible],
                                   [gloc_can_end_within_visibility_window(contact_graph,gloc_duration)],
                                   [gloc_can_end_before_next_comm(contact_graph,gloc_duration)]])
            case "RADIO_GLOC":
                self.rules.extend([[gloc_only_when_sat_is_visible],
                                   [gloc_can_end_within_visibility_window(contact_graph,gloc_duration)],
                                   [gloc_can_end_before_next_comm(contact_graph,gloc_duration)]])
            case "VISUAL_GLOC":
                self.rules.extend([[gloc_or_measurement]])
            case _ :
                raise ValueError(f"Policy type {policy_type} not recognized")
        self.rules.extend([ [wait_to_wakeup_before_comm],
                            [drive_in_sequence],
                            [drive_only_if_not_driving_already],
                            [measure_only_if_not_measuring_already],
                            [recharge_only_if_not_recharging_already],
                            [stop_to_communicate,stop_if_no_energy,stop_to_gloc,stop_to_measure],
                            [stop_driving_if_no_energy],
                            [drive_only_if_not_gloc_or_measurement],
                            [dont_measure_while_resetting],
                            [drive_only_if_not_recharge_to_max],
                            [drive_only_if_soc_above_threshold]])
```

Figure 4: Example of policy specification in our simulation environment.

friendly for software engineers. This is important if we intend for engineers to write models themselves. An alternative choice would be to use a formal modeling language such as Lean, PVS, TLA+, Event-B, or Spin. The advantage of using formal methods is that a model can be verified for all possible execution scenarios, or at least many more than we can achieve by executing a Python program. Our plan is to pursue a formal approach beyond just testing. We consider the pursuit of one or both of the following two alternative approaches: (i) supporting formal verification of Python programs and (ii) the creation of a formal model in a proper formal specification language.

Formal verification of Python programs— The simulation model introduced in Section 5 is implemented as a Python program. The ability to formally verify such programs would provide a straightforward path to the formal analysis of autonomous systems at the lv_c level. One approach [38] consists of translating Python into C or C++ to leverage the ESBMC bounded model checker [39]. ESBMC translates a C or C++ program into an SMT problem by unrolling loops to a specified depth k and then searching for property violations across all possible "executions" up to that depth. This offers the advantage of a user-friendly language for writing models while supporting both simulation and formal verification. Whether this approach is computationally viable remains to be investigated.

Using a Proper Formal Language—The alternative is to use a formal specification language. Numerous such languages and tools exist [40], which can be categorized as follows: *Theorem provers* offer very expressive higher-order logic specification languages. However, they require users to carry out proofs interactively with the system – for example, by providing loop invariants. Their advantage lies in the flexible mathematical formalisms, making them suitable for defining complete theories and concepts. Examples of theorem provers include Lean, Coq, PVS, and Isabelle/HOL (we refer to [40] for references to formal methods tools). *Model checkers* usually offer less expressive specification languages but support automated proofs. Automated proofs are of course advantageous, especially if users are engineers with little training in formal methods and under time pressure. Examples of model checkers include TLA+, Spin, nuSMV, and FDR4. *Formal programming languages* are a special group of formalisms which are, or resemble, programming languages and therefore should be more accessible to engineers. While most of these are effectively theorem provers, requiring manual proofs (e.g., Dafny), some are based on model checking, including Dezyne, Quint, and P.

We are still in the process of selecting the most suitable approach for our applications. There is value in formalizing an architecture in an expressive formal language (as supported by theorem provers), in automated verification (as supported by model checkers), and in a formalism close to a programming language which is accessible yet expressive. Our initial attempt to use Python as a modeling language and support it with formal verification is an attempt to address these advantages.

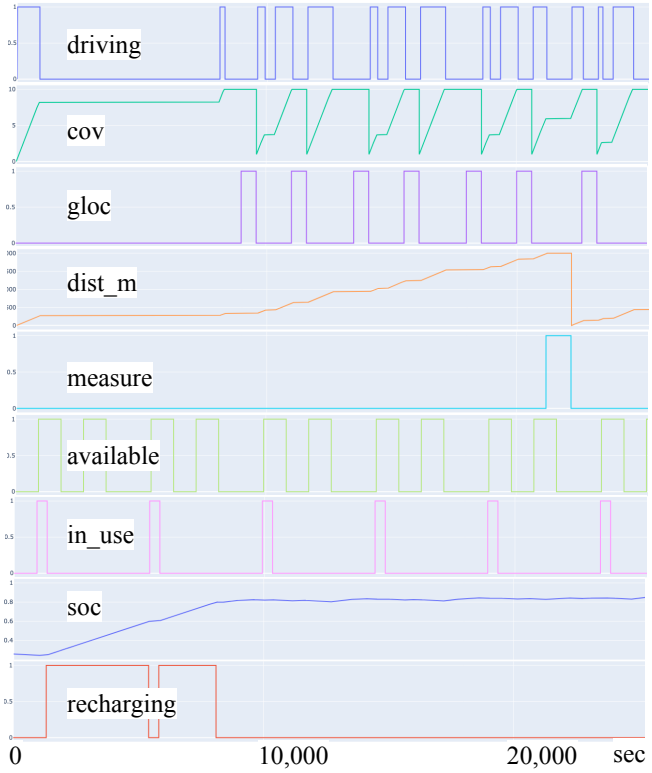

Figure 5: Results obtained by simulating the lunar rover case study described in Section 4 (Lunar rover example).

.

6. TEST GENERATION

Once a set of requirements at lv_i has been identified, a V&V activity is required to check that they are satisfied at level lv_{i+1} – a special case is to check that the flight software implements lv_p requirements. When an executable model for lv_{i+1} is available, then V&V can be conducted by testing the system under many possible scenarios, and verifying that each test results in a trace of values for the state variables that does not violate lv_i requirements. Testing for autonomous systems has been investigated in academic works [41, 42, 43], and widely used by the automotive industry [44]. This approach requires sampling algorithms that can (intelligently) generate these varied tests. Due to the complexity of typical autonomous systems, namely the number of state variables at all levels, the number of behaviors, and the uncertainty coming from limited knowledge of the environment, the number of possible scenarios to evaluate becomes too large to be fully explored. Thus, the sampling algorithms must work efficiently to identify the most important subset of tests to execute. The importance of each test is a major consideration in this work. Tests that are similar take time and budget resources away from more interesting tests. Ideally, the system is tested under *enough* scenarios that are widely distributed across the entire state space so that mission stakeholders can gain confidence that the SUD can meet the requirements in all the expected conditions. Accordingly, another major focus of this work is to determine when "enough" tests have been executed. Since exhaustive testing is generally not possible, approaches must be defined to *estimate* when the number of (successful) tests is sufficient such that that the implementation of the autonomous system can be trusted to work properly. This concept of coverage, or the variance in both the test inputs and the test outputs that provides insight

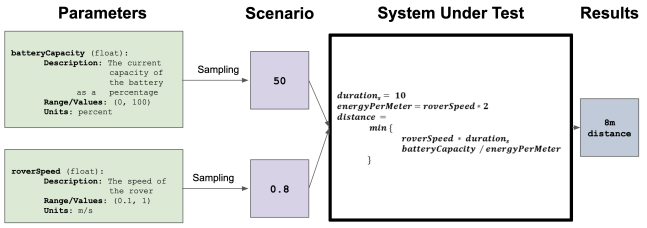

Figure 6: An example of selecting parameters, sampling values for the parameters to produce a scenario, and then providing the scenario to the system under test which in turn produces some metrics, in this case the total distance the rover drove given the scenario.

on when enough testing has been performed, is discussed more in Section 6 (Testing Coverage).

The test generation framework as described below in Section 6 (Architecture Overview), has been architected for the following purposes:

- Large-scale *data generation* from system runs for the purpose of analysis or learning. The framework is used to generate and execute tests with no feedback to direct an intelligent search for edge cases.
- Finding the solution of *optimization problems*. In this case, the framework leverages feedback on how well the system under test performed in previous tests. This is useful for tasks such as tuning controller parameters to achieve optimal performance, where the optimization algorithm identifies the best controller parameters.
- The final and arguably the most interesting use case is to *identify requirement violations and compute an assurance metric* that defines the probability of a requirement violation. Identifying requirement violations may either be performed as simply identifying a violation and then stopping the test generation until a developer has fixed the identified problem, or it may be used to compute the performance bounds of the system under test. The latter involves identifying (all) the regions of the parameter space where the autonomous system violates the requirements and not just the first violation.

Figure 6 is a visual depiction of the nomenclature that we will use in this section:

- **Parameter**: A variable in the system that can take on values within a given range. There are engineering parameters (i.e., battery capacity, downlink rate), science parameters (i.e., whether a bio-signature will be positive, instrument calibration), or environment parameters (i.e., average rock size, tortuosity of the terrain).
- **Sampled / grounded parameter**: A parameter that has been assigned a particular value from its domain. For example, setting the value of a "battery capacity" parameter to 60% state of charge.
- **Sampling**: The act of selecting a value for a given parameter such that the parameter is now a sampled / grounded parameter.
- **Scenario**: The set of sampled parameters provided as input to the system under test. Note that "scenario" and "test" can be used interchangeably in this paper, as generating a test for the system-level autonomy is the same as generating a set of sampled parameters to provide as input to the system under test.

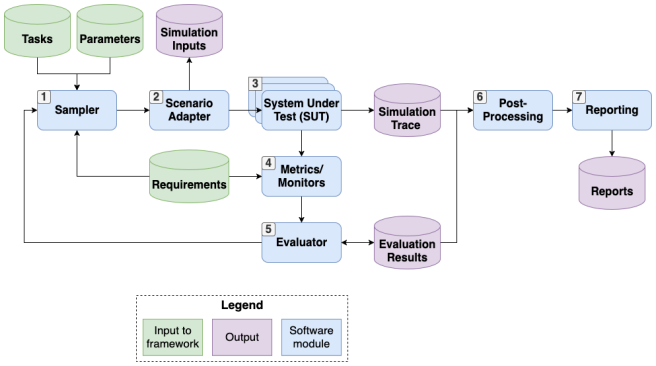

Figure 7: An overview of the test generation architecture, generating scenarios based on the provided parameters, running the simulation with the scenarios, and then evaluating the results of the simulation as part of a feedback cycle.

Framework Design Goals

The test generation framework is meant to support several different design goals.

1. **Modularity**: Each component of the framework must be independently modifiable and extensible. This is to ensure sufficient flexibility in the actual generation and evaluation of the tests. For example, the architecture should allow multiple sampling algorithms, optimization methods, evaluation criteria, and reporting methods. The framework should also be agnostic to the implementation of the underlying system under test (such as planning algorithms or inter-process communication mechanisms).
2. **Scalability**: Due to the large state space of possible scenarios, it is desirable to have a framework that can easily scale and support distributed (parallel) test executions and analysis.
3. **Reproducibility**: It should be possible to repeat the execution of any test that is generated. This is especially important during development because it allows engineers to correct software that leads to a requirement violation and then re-evaluate a given test to ensure that it is now passing. This means saving scenarios and execution history in logs for the ability to be replayed as well as analyzed.
4. **Real-time monitoring and reporting**: Especially as the number of tests that are run increases, it is desirable to track the progress of the testing in real-time as well as generate reports that summarize both the interim and final results.
5. **Ease of integration and use**: To avoid impacting the productivity of the software developers, the framework should be frictionless and intuitive to use, with robust error handling and feedback messages. Major modifications to the software under test should not be required.

Architecture Overview

The design goals described in the previous section guided the design of the architecture of the test generation framework shown in Figure 7. We describe each component next.

Parameter Selection. The first step in automating test generation is to identify the set of parameters that capture all scenarios of interest. It may be beneficial to have a variety of parameters, for example the starting battery capacity, the location of obstacles, or a binary value that represents whether a thermal fault should be triggered. The selection of parameters can be done by SMEs, can be derived from models (e.g., the model used in Section 5), or can be iterative (the results of a testing campaign may identify new parameters that should be included or parameters that are irrelevant and should be removed from the global set of parameters). Selecting the right parameters is non-trivial but essential to the assurance process as the SUD will be assured only under the parameters that were explicitly varied.

Sampler Component. The sampler component takes in the set of parameters and grounds each parameter to produce a scenario that is used to evaluate the system under test (SUT). The sampler component is designed to support a variety of sampling algorithms either for exploratory research or to allow users to select the algorithms that work best for their purpose.

The sampler component supports constraints on parameters that are typically needed to ensure realistic inputs. For example, we may want to constrain the initial position of a rover to be outside of an obstacle. Any scenario that violates parameter constraints is rejected.

The sampler component also supports warm starts, which are scenarios that SMEs have manually defined. This feature is useful to guarantee that certain scenarios are always evaluated (because of interest to SMEs), and to enable a continuous integration pipeline, where the same tests can be run to ensure that the latest code changes do not break the software or cause it to violate a requirement.

More details about the supported sampling algorithms are provided in Section 6 (Sampling Algorithms).

Scenario Adapter Component. The scenario adapter component takes the scenario as input and may optionally modify it into a format that can be used by the system under test. For example, this may mean saving the scenario to a YAML file that is loaded by the SUT, setting ROS parameters, or publishing ROS messages. While this component may be implemented by the specific SUT, the test generation framework provides several tools to assist in this effort. For reproducibility, all scenarios are saved to JSON files.

A frequently used feature is the ability to define a configuration template file for the SUT. The template contains a set of parameters $P_{SUT} \supset P_S$, where P_S is the set of parameters that defines a scenario. The scenario adapter creates copies of the template and replaces the values of P_S according to the current scenario before running the SUT.

System Under Test Component. The system under test component is the SUD together with its environment. It should accept a configuration and it should generate outputs to be evaluated. Ideally, this component supports parallel execution so that multiple executions can be performed at the same time.

Monitoring Component. The monitoring component is used to evaluate the performance of the autonomous system. In particular, this is used to ensure adherence to all requirements. This component may be run either in parallel with the SUT (on-line monitoring), or after the SUT has finished running (off-line monitoring of the generated log files). In addition to returning the number of times each requirement was violated, the monitors may also be used to extract metrics from the test run, such as the distance that a rover drove or the time it took to complete a mission. These metrics may be used in post-processing to analyze the behavior of the SUT as well as to provide feedback to the sampling algorithm on which scenario to generate next. The metrics may show undesirable performance for a specific test, which might lead to a new requirement that constrains the behaviors to be more desirable. The monitoring technology is described in more detail in Section 7 (Monitoring).

Evaluator Component. The evaluator component is an optional component that will take the feedback provided from the monitoring component and use that information to update

the sampling algorithm. Typically, the sampler component and evaluator component are implemented together. For example, most of the sampling algorithms involve a sampler component that generates scenarios, and an evaluator component that updates some internal or surrogate model (such as a Gaussian process in the case of Bayesian Optimization or the current populations for a genetic algorithm).

The feedback that is provided to the evaluator is typically an objective function that is used in tandem with an optimization algorithm. The exact objective function that is being optimized is provided by the user, but it is typically either the metrics or the number of requirement violations returned by the monitoring component. If the selected algorithm supports multi-objective optimization, the metrics or requirement violations can be provided as a list of results to the evaluator, otherwise the results must be aggregated into a single metric.

Post-processing and Reporting Components. Once all tests have been executed, the results will be post-processed by the post-processing component into a format that can be used by the reporting component to generate reports for the user.

The post-processing currently involves aggregating data from the runs and then computing information such as the total time taken by each test.

It will also compute some properties such as the permutation importance and partial dependence of all parameters in order to determine how the results of the tests are influenced by the different parameters. These results can be used to inform users which parameters have little to no impact on the final results and as such can likely be removed from the set of parameters so as to lessen the search complexity. Figure 8 shows an example of this, where the test generation was used to minimize a score defined as the time it took a simulated rover to reach all of its waypoints within a fixed timeframe (all waypoints must be reached by the end of the simulated mission). Several parameters were identified that had the largest impact on the time to complete the mission, namely the speed of the rover $\bar{v}$ as well as the error tolerances in the navigation and system level autonomy modules that determine how far from a waypoint the rover can be before the waypoint is considered reached, denoted th_n and th_s, respectively. Other parameters such as the frequency at which the system level autonomy and navigator controller ran had much less of an impact on the time to reach all waypoints. In Figure 8, the permutation importance plot identifies these parameters as the most essential given the objective function. The partial dependence plot shows how increasing the speed and system level autonomy thresholds while decreasing the navigator thresholds produced the fastest mission durations. The dual partial dependence plot shows that these three parameters were highly correlated with one another. The parameter correlations plot provides similar information to the partial and dual partial dependence plots in that the parameters are highly correlated with one another. The overall goal of these plots is to provide information to the user regarding the influence each parameter has on the simulation. While not discussed in this paper, this information can also be used to ensure that appropriate parameters are provided to the test generation to generate tests, and that parameters with little to no influence on the results of the simulation can be removed from consideration, thus reducing the size of the parameter space.

For each run, HTML reports are automatically generated to provide sufficient information to the users to understand what occurred during the tests, as well as global reports that summarize findings across all tests. These reports are essential for debugging purposes such as notifying developers exactly when a requirement was violated and providing some context on how it was violated. For example, Figure 9

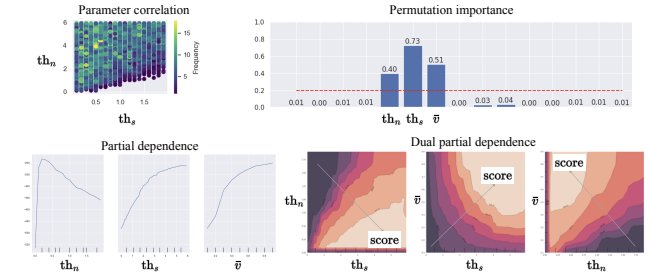

Figure 8: Some examples of analysis that is automatically performed by the test generation such as showing which (pairs of) parameters (and their values) had the greatest impact given an objective function. The goal is to gain some understanding in how the outputs of the system under test varies as a function of the provided parameters.

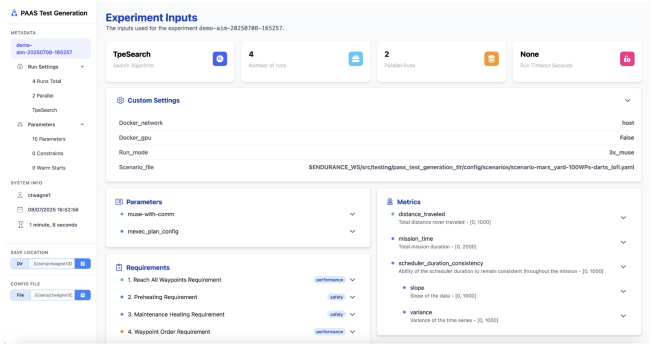

Figure 9: An example of a generated report showing the inputs provided to the test generation.

displays the inputs provided to test generation given the SUT, such as the parameters the system under test will take as input, the requirements that will be used to evaluate the system under test, the metrics the system under test will produce, the sampling algorithm used to generate tests, as well as the total number of tests generated. Figure 10 shows another dashboard that presents the results of each test such as whether the test succeeded, failed, or produced an error, the score of the test given the objective function, the scenario used in the test, as well as the numeric values for each metric returned by the system under test.

The reporting component also includes features such as the ability to send email or push notifications to notify users when a set of tests has completed so that the user does not have to keep checking the progress of the tests.

Handling Large-Scale Runs

In order to support generating and executing large numbers of tests, several features were added to the test generation framework. The first is to include the Ray library [45] which makes it easy to distribute the workload of executing several runs in parallel given the current hardware limitations.

Automatic test generation generates a large quantity of data which may present several challenges:

- *Storage requirements*. To address this challenge, a data retention evaluator component was implemented to determine whether the data from a given test should be stored or deleted. For example, it may be acceptable to delete the execution trace for a test that produced no requirement violations and

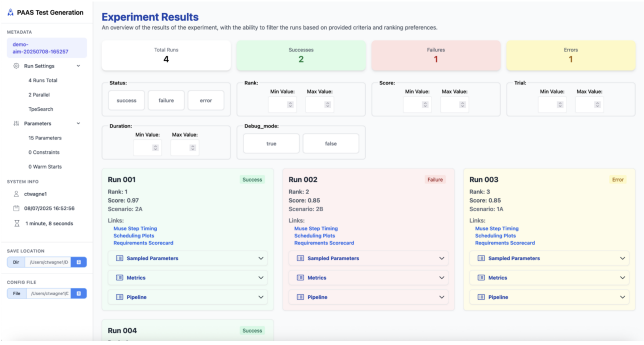

Figure 10: An example of a generated report showing some of the outputs produced by both the test generation and the system under test.

no interesting metrics. However, tests with requirement violations should be kept for further analysis. The ability to re-run specific tests can be leveraged in cases where users decide to inspect even uninteresting ones.

- *Prioritization of manual analysis*. A ranking algorithm is needed to prioritize which tests to inspect first. The test generation framework supports several multi criteria decision making (MCDM) algorithms such as TOPSIS and Penalty Boundary Intersection (PBI) that allow users to weight (multiple) criteria to order the tests, such as ordering tests based on the number of requirement violations and the number of failed mission objectives.
- *Debugging*. Debugging can also be a challenge at scale, so several tools were developed to assist engineers in finding problems. The first is defining a lightweight pipeline software service that allows users to architect their tests as pipelines as shown in Figure 11. These pipelines, which are executed for every test, can be used to better manage the way the system under test interacts with the test generation framework. As shown in Figure 7, users of the test generation framework must define components such as the scenario adapter (which in the example diagram involves taking the inputs from the test generation and saving them to a YAML file), the system under test (which in the example diagram involves several steps such as generating shell scripts to start the simulation referred to as TLR, running the shell script to start a Docker volume, extracting some metrics from the log files of the simulation, removing or compressing data, generating visualizations, in addition to killing the Docker volume once the simulation has ended), the monitoring component where the monitors are run to evaluate how the system under test performed and whether it met all the requirements, as well as the evaluator component which in the example diagram is the Metric Evaluation step which generates metrics that are provided to the test generation sampling algorithm. Because the test implementations typically require many complex steps, an explicit representation of a pipeline makes it significantly easier for developers to structure their code as separate (hierarchical) steps that are executed. Pipeline visualization, which is automatically generated from its definition, also makes it easier to explain to others the steps that are executed in each test as well as identify where problems occur in a given test. For example, Figure 11 shows an example of pipeline execution showing that the relevant log files could not be loaded to produce metrics, which can be traced back to a simulation failure. The pipeline implementation is also meant to ensure that all errors are handled gracefully so that if a particular test results in an error, other tests are not impacted. Another helpful feature of the pipeline is the ability to re-run specific steps of the pipeline. For example, a developer might fix a bug in how log files are processed and then re-run only that step; there may not be a need to re-execute the entire test itself but just process the existing files with the updated code.
- *Automatic visualization*. Finally, automating the creation of visualizations, in particular by rendering plots in Jupyter notebooks for each test, can be of tremendous help. This allows users to visualize plots and tables as well as re-run cells or modify the code used to generate the plots or tables.

Sampling Algorithms

The test generation framework supports a variety of sampling algorithms. Thanks to the modular nature of the framework, the effort required to add sampling algorithms from external libraries such as Optuna [46] is minimal. The types of supported sampling algorithms can be categorized as uninformed, informed, resource-aware, and coverage-guided.

Uninformed algorithms generate scenarios without taking into account the result of executing previous tests (no feedback). Some of the implemented algorithms include grid search, Monte Carlo sampling [10], and combinatorial (t-way) testing using the Advanced Combinatorial Testing System (ACTS) [47] that systematically generates a minimal set of test cases that cover all combinations of input parameters up to a specified strength (ie all pairs, triples).

Informed algorithms are connected in a feedback loop with the SUT and refine their search based on the result of running a test. Some of the implemented algorithms include Bayesian Optimization [48], Genetic Algorithms [10], CMA-ES sampling [49], and Simulated Annealing [10]. Several of these algorithms also support multi-objective optimization, which can allow users to explore the Pareto frontier of multiple metrics of interest.

Resource-aware algorithms are informed algorithms that also assign a certain resource budget to tests. The goal is to assign minimal resources to each test early on to quickly evaluate performance and decide whether to devote more resources for a later test. These algorithms are common in hyperparameter optimization for machine learning where the resource might be the amount of computational power and training data provided to a parameterized neural network; neural networks that perform well early on will be given more resources to improve their performance while networks that fail early on will be given no future resources. In our case, the resource is typically the amount of simulation (mission) time. Each test starts with a small amount of simulation time to see if requirements are violated early on, with the assumption that it is not needed to keep simulating a mission that has already violated a requirement. For tests that do not have any requirement violations, they will later be run for a longer period of time to check if they still do not violate any requirements. Some of the implemented algorithms include successive halving [50] and hyperband optimization [51].

The final category is the coverage-focused algorithms. These algorithms attempt to maximize coverage of the parameter space rather than optimizing an objective function. The concept of coverage, as well as maximizing coverage is further described in Section 6 (Testing Coverage).

Testing Coverage

A central challenge in test generation (and V&V in general), is determining when the system has been tested "enough"

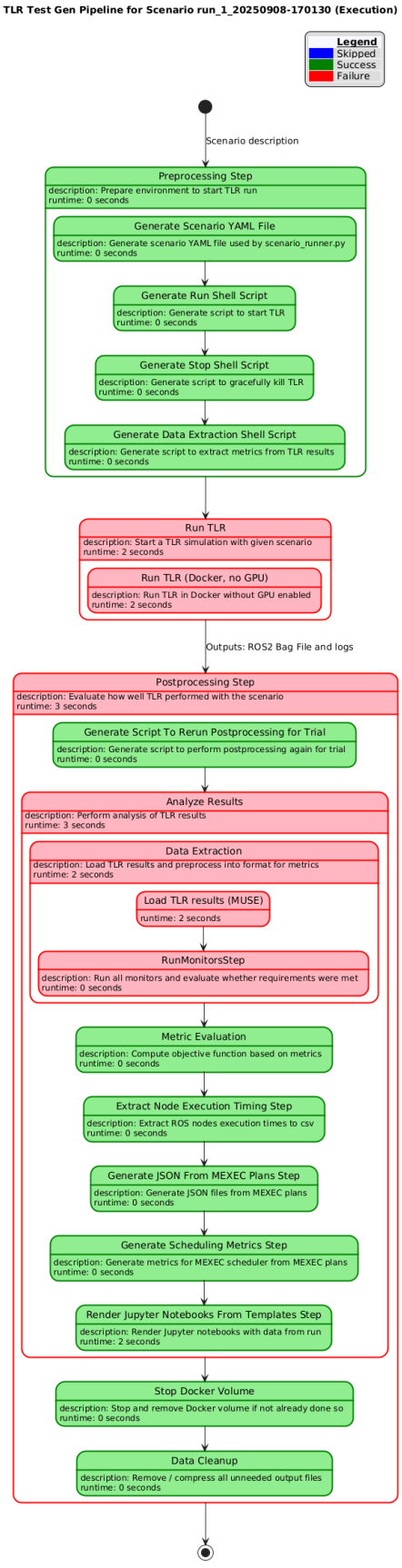

Figure 11: An example pipeline diagram automatically generated from the test generation implementation. This shows how some of the test generation architecture components were implemented for a given system under test (referred to as TLR), such as the scenario adapter, system under test, monitoring, and evaluator components of the architecture.

to provide sufficient confidence that the SUT will work as intended (recall that we exclude the possibility of exhaustive testing for autonomous systems) [52], [53]. Sufficiency of the set of tests is synonymous with the residual uncertainty about the system's behavior. Given a budget for the level of effort spent in testing the SUT, the set of tests must therefore be selected in such a way as to maximize the diversity of the test inputs and outputs to reduce uncertainty. We refer to the relative reduction in uncertainty as *coverage*. More specifically, *parameter space coverage* refers to the diversity in test inputs, while *requirement coverage* and *behavior coverage* refer to the diversity of the observed test outputs. Ideally, the sampling algorithm maximizes each of these coverage metrics so that the system under test is sufficiently evaluated.

Coverage can be quantitatively calculated as the normalized reduction in uncertainty between a prior belief about system behavior and the posterior belief after testing:

$$Coverage = 1 - \frac{U_{\text{Posterior}}}{U_{\text{Prior}}} \tag{1}$$

where U_{x} is an estimate of the degree of uncertainty in the ability of the system to satisfy the requirements. Examples of U_{x} include posterior variance, entropy, or other dispersion measures of the evaluation output. Note that this general definition is agnostic to the specific modeling framework such as whether Bayesian networks, Gaussian processes, or Beta-Binomial models are used to define prior and posterior uncertainty. Intuitively, if no testing has been conducted, the posterior equals the prior and therefore the coverage is zero. If testing fully resolves uncertainty (i.e., exhaustive testing), coverage is one. Otherwise, partial reductions in uncertainty yield coverage values strictly between zero and one.

It is important to note that this formulation of coverage is meant to be implementation specific and so does not allow comparisons between different systems. Furthermore, this coverage is computed based on the provided input parameters. A coverage score of one, meaning that all uncertainty is resolved, does not guarantee that the system under test will work as expected for all cases, but instead that it will work as expected given the parameters that are being varied. If there are other important parameters that are not included, this coverage metric will likely not capture the full scope of the system.

Parameter Space Coverage

The goal of parameter space coverage is to maximize the coverage of the inputs provided to the system under test to ensure that scenarios that lead to diverse, failing, or unexpected outcomes are in fact evaluated. Parameters can be both continuous and discrete, which impacts the selection of the sampling algorithms.

An example of an algorithm that attempts to maximize parameter space coverage is the Gower Max-Min. This algorithm is uninformed (does not receive the current coverage value as feedback), but aims at generating an even distribution of samples across the parameter space. Gower Max-Min sampling uses Gower's distance to maximize the minimum distance between all scenarios. Gower's distance [54] is a metric that can compute the distance between a mix of both continuous and discrete parameters, something with which other popular algorithms such as Latin hypercube sampling often struggle. The distance of the set of parameters is the sum of the distance over each parameter in the set.

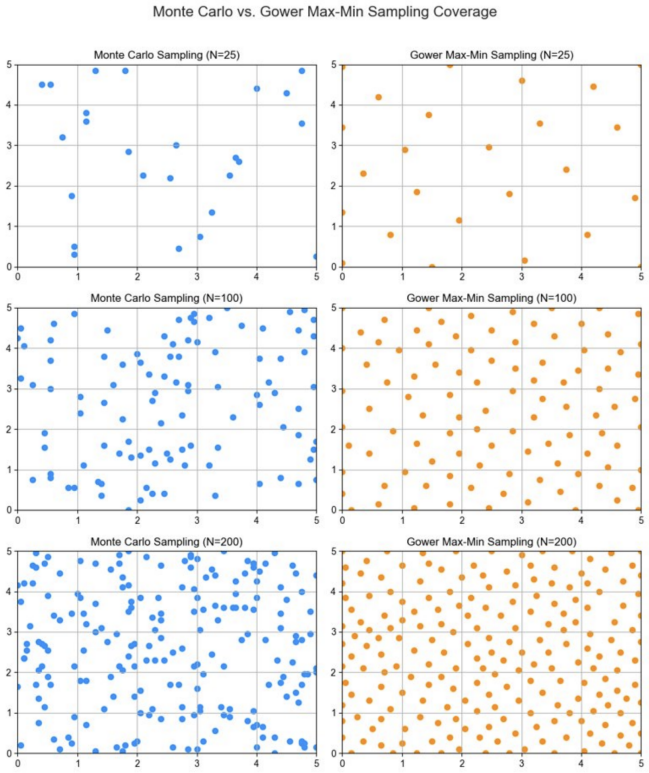

Figure 12: A comparison of Gower Max-Min sampling with Monte Carlo sampling. As shown in the diagram with a 2d (two parameter) example, Monte Carlo sampling tends to produce clusters of values while Gower Max-Min is much more evenly distributed across the parameter space. This ensures more event coverage of the entire parameter space.

The distance between continuous parameters is simply their difference, and the distance between discrete parameters is zero if the two parameters are equal and one otherwise. This distance is computed for all pairs of scenarios. The first scenario is randomly chosen, and then all subsequent scenarios are greedily selected as having the greatest distance from all previously sampled scenarios. As shown in Figure 12, this algorithm can achieve an even sampling distribution of the parameter space relative to commonly used algorithms like Monte Carlo sampling.

Another algorithm that is not only coverage-focused but also coverage-aware is referred to as Hierarchical Uncertainty Coverage (HUC). HUC probabilistically models the performance of the SUT across the parameter space, similar to Bayesian optimization. It then generates new samples based on where the algorithm has the greatest uncertainty regarding the expected performance of the SUT. HUC uses the coverage definition in Equation (1) and provides a concrete representation of the prior and posterior uncertainty. It partitions the input parameter space into a tree of leaf regions. Each leaf maintains a posterior distribution over requirement satisfaction outcomes. Each leaf ℓ computes:

- a probability mass w_ℓ, such as a weight provided by SMEs to bias the sampling towards different regions of the parameter space. Typically however, it is equal to the volume of the partition that the leaf represents.
- a posterior variance $\mathrm{Var}_\ell[p]$ representing uncertainty in requirement satisfaction probability p within that region. For binary (Bernoulli) requirements, a Beta-Binomial model is typically used.
- a prior variance $\mathrm{Var}_{\text{prior}}$ representing the *initial* uncertainty in requirement satisfaction probability p within that region. For binary requirements, a Beta-Binomial model $Beta(1,1)$ is typically used where the prior weights of the model are low such that the variance will be large.
- a finite population correction term Fpc_ℓ which keeps track of the number of scenarios within the leaf partition that have been executed, divided by the total number of possible scenarios that could be executed within the leaf partition.

Coverage under HUC is then computed as:

$$\text{Coverage} \;=\; 1 - \frac{U_{\text{Posterior}}}{U_{\text{Prior}}} \;=\; 1 - \frac{\sum_{\ell\in\mathcal{L}} w_\ell \mathrm{Var}_\ell[p]\mathrm{Fpc}_\ell}{\sum_{\ell\in\mathcal{L}} w_\ell \mathrm{Var}_{\text{prior}}[p]}, \tag{2}$$

where $\mathcal{L}$ is the set of leaf regions. Equation (2) aggregates variance across the hierarchy, weighting the variance by the probability mass of each region. As testing proceeds, the posterior variance in each leaf decreases, and thus overall coverage increases.

Adaptive Refinement Strategy—To maximize coverage efficiently, HUC adaptively refines the partition tree. Leaf nodes with high uncertainty and large probability mass are split into smaller subregions. Sampling is concentrated in leaves that maximize a score such as

$$S_\ell = w_\ell \cdot \frac{\mathrm{Var}_\ell[p]}{\sqrt{n_\ell + 1}}, \tag{3}$$

where n_ℓ is the number of samples already drawn in leaf ℓ. This bias balances exploration of under-sampled leaves with exploitation of regions where uncertainty remains high. Over time, the hierarchy resolves into fine partitions only where necessary, leaving uninformative regions coarse. In this way, the sampling is biased towards areas of the parameter space that lead to diverse sets of outcomes rather than spending time in regions of the parameter space for which all outcomes are the same. This process of splitting leaves into smaller partitions and computing the uncertainty is depicted in Figure 13.

Note that although HUC is described above in terms of Beta-Binomial posteriors, the same coverage algorithm can be instantiated with other models. For example, Gaussian processes can be used to represent continuous uncertainty, with coverage measured by the reduction in average posterior variance across the parameter space. It is also possible to try alternative methods such as information gain as the metric for which leaf nodes are split into multiple smaller partitions. This is currently under investigation as future work.

Requirements Coverage

Requirements coverage assesses whether all conditions expressed in a requirement have been tested under all relevant conditions. The metrics and algorithms to achieve requirement coverage depend on the language used to express requirements. For requirements expressed as state machines (such as the monitors described in Section 7 that the test generation uses), or temporal logic formulas that can be converted to finite state automata, requirement coverage involves ensuring that all critical states and transitions are visited. For example, if a requirement specifies that a rover must choose between transmitting collected data or acquiring a new sample, both branches of this decision must be exercised during

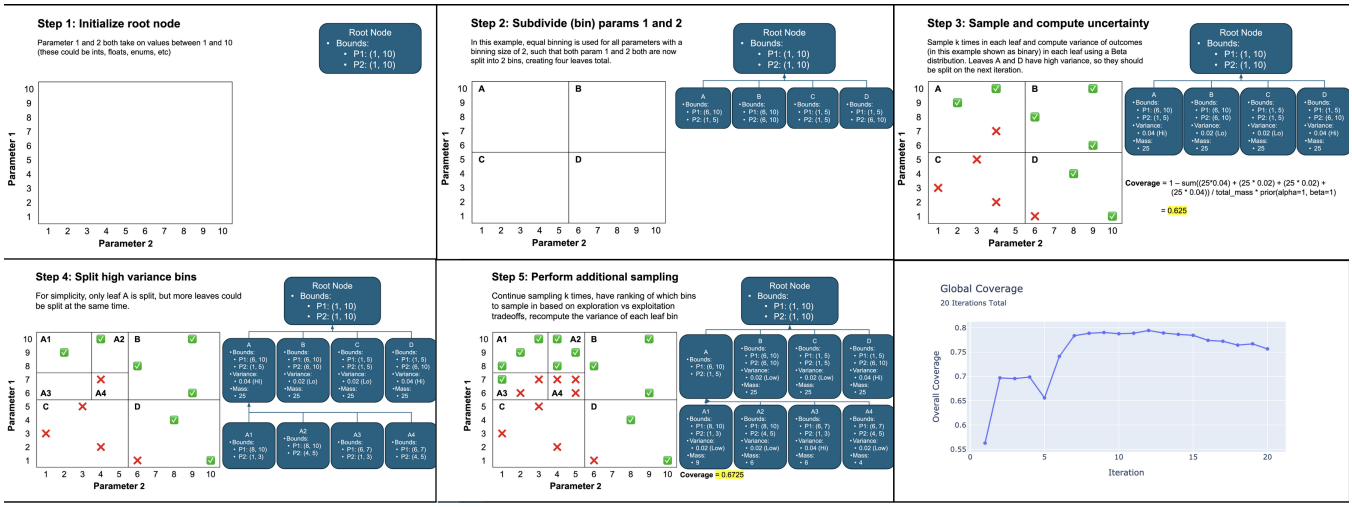

Figure 13: An overview of hierarchical uncertainty coverage and how it recursively subdivides the the parameter space into bins (leaf nodes) based on the amount of variance computed within each bin. This allows a global coverage metric to be computed which is the weighted sum of uncertainty computed in each bin divided by the prior uncertainty, thus computing the overall reduction in uncertainty after running the tests.

testing to confirm correct handling. Requirements coverage therefore helps reveal untested logical pathways or state transitions and complements parameter space exploration. Figure 14 depicts an example of the requirement coverage, formalized as the likelihood of violating a requirement.

Behavior Space Coverage

Behavior coverage captures whether the full set of behaviors that can be generated by the system has been observed and validated. As with requirement coverage, the metrics and techniques used for behavior space coverage depend on the representation of the desired behaviors. For task-based representations, such as hierarchical task networks (HTNs)[30], behavior coverage involves ensuring that all tasks and constraints are encountered at least once. This is very similar to software code coverage where the goal is not only to verify that all paths in the code have been exercised, but also to identify any unused or unevaluated behaviors that could introduce latent errors during operations. It is undesirable to have task instances in a task network whose execution has never been triggered during testing, but that could be triggered during a space mission. A task that has never been scheduled for execution indicates that the task is not needed and should be removed, or that additional tests should be generated to expand the set of behaviors that have been verified against requirements. Figure 15 shows an example report that depicts an example behavior coverage of a SUT.

7. Monitoring

The use of monitoring is a significant aspect of our assurance process. The overall goal of monitoring is to automate the evaluation of what occurred in a given test. For autonomous systems, where failures may depend on complex sequences of events across many state variables, the traditional ad-hoc approach that relies on SMEs inspecting logs with the support of data visualization tools is slow, tedious, and error prone. It does not scale to thousands or millions of tests that might be needed to achieve full coverage.

We advocate for the use of temporal monitors which relate

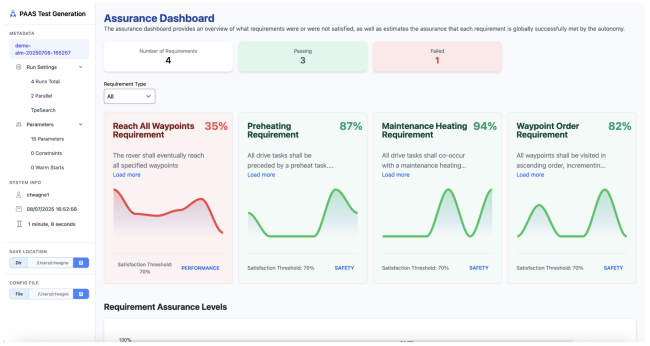

Figure 14: A report showing the assurance values of each requirement, which represent how confident the user should be that each requirement is met by the system level autonomy (note this is a toy example and not realistic values).

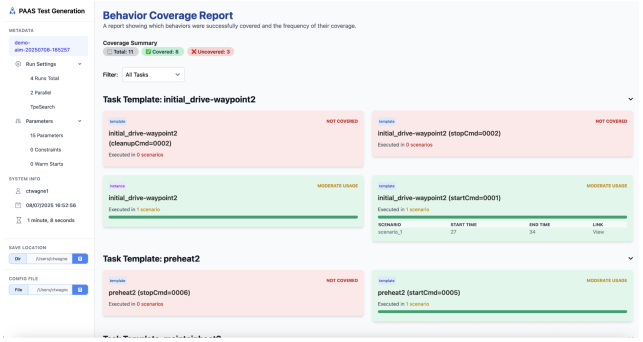

Figure 15: A behavior coverage report produced by the test generation that shows which of the behaviors the system level autonomy can perform were actually executed (note this is a toy example and not realistic values).

```python
class PreheatMonitor(pyc.Monitor):
    def transition(self, event):
        match event:
            case StartPreheatCommand():
                return self.PreheatState()

            case CancelPreheatCommand() if not self.PreheatState():
                error_message = "Tried to cancel preheat but not preheating"
                return pyc.error(error_message)

            case DriveToWPCommand(waypoint_id=waypoint_id) if (
                not self.WaitingForDriveCommandState()
            ):
                error_message = f"Drove to {waypoint_id} without preheating first"
                return pyc.error(error_message)

    @pyc.data
    class PreheatState(pyc.HotState):
        def transition(self, event):
            match event:
                case PreheatCompletion(success=True):
                    print("Successfully finished preheating")
                    return PreheatMonitor.WaitingForDriveCommandState()

                case PreheatCompletion(success=False):
                    return pyc.ok

                case CancelPreheatCommand():
                    return pyc.ok

    @pyc.data
    class WaitingForDriveCommandState(pyc.HotState):
        def transition(self, event):
            match event:
                case DriveToWPCommand(waypoint_id=waypoint_id):
                    print(f"Driving to waypoint {waypoint_id}")
                    return pyc.ok
```

Figure 16: Example of a PyContract monitor.

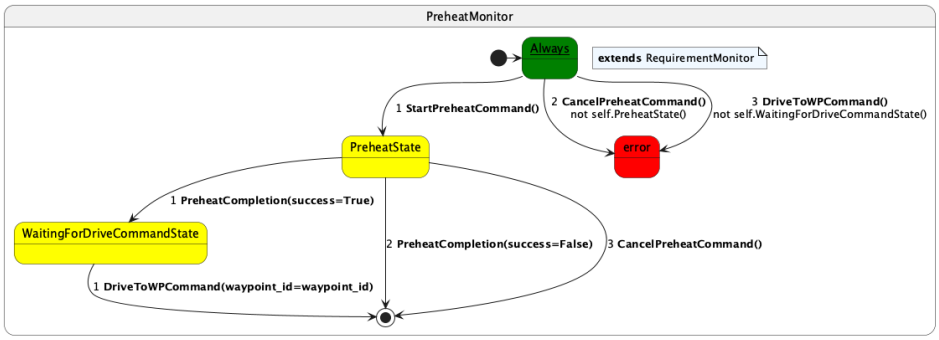

Figure 17: Visualization of the monitor in Figure 16.

events occurring at different time points. An example of such a property is: "*after a successful preheat, a drive-to-waypoint should follow, unless the preheat fails or is cancelled*". These properties should capture behavioral preferences at an abstract enough level to allow an autonomous system to make different decisions depending on the situation. The process outlined in Section 3, which advocates for permissive policies, should result in requirements that allow for such flexibility. Temporal properties can be written in several forms and languages, but an automaton-based description may be easier to work with for engineers who might not be experts in formal methods. We describe a rule-based Python library for implementing automaton-flavored monitors.

The PyContract Monitor Library

We use the Python library PyContract [55, 56] as a monitoring framework. The framework is flexible enough to accommodate a wide range of specifications over sequences of events that can be of any type allowed by the Python type system. The stream of events can be submitted to a monitor via method calls, but we have also developed an interface to ROS2 topics. PyContract offers features (classes, methods, functions, and constants) for defining temporal monitors. The set of monitors to be deployed for a given application comes from requirements, but additional monitors are usually added in an iterative process. Anomalous behaviors can still be present due to missing requirements. The identification of these behaviors is still a manual process.

As an example, Figure 16 shows a monitor verifying that pre-heating of our example rover described in Section 4 is done correctly before driving. More specifically, the monitor checks that the following requirements are met:

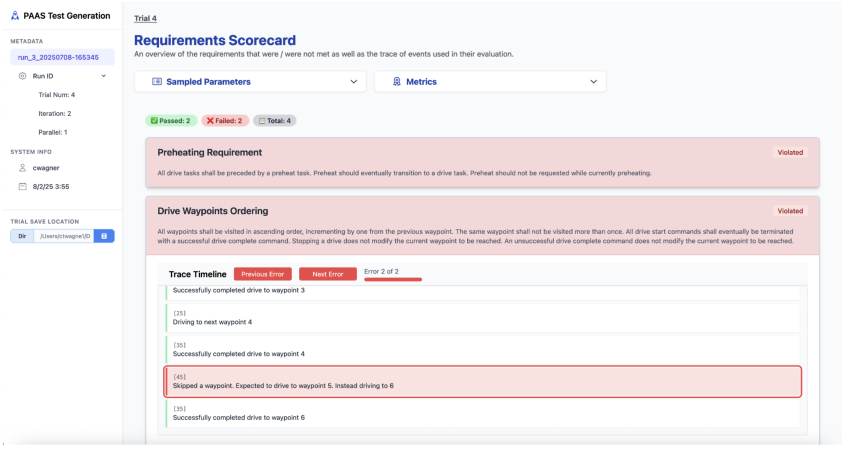

Figure 18: Monitor status report for a single test

1. *A drive task shall be preceded by a successful preheat task*
2. *A preheat task shall eventually transition to a drive task*
3. *A cancel preheat task shall not be requested if the rover is not preheating*.

A monitor in PyContract is defined as a class derived from the `Monitor` class that accepts events. The `transition` method (Line 2-15) is invoked on all events submitted to the monitor. This monitor listens for commands to start preheating (`StartPreheatCommand`), stop preheating (`CancelPreheatCommand`), as well as to start driving to a waypoint (`DriveToWPCommand`). A monitor is a form of state machine. The monitor uses two separate states: the `PreheatState` (Lines 17-29), which is entered (made active) when the system starts preheating (Line 5), and the `WaitingForDriveCommandState` (Lines 31-37), which is entered when preheating has successfully completed (Line 23) and the system is now waiting to start driving to the next waypoint. These two state classes are derived from `HotState` (Lines 18 and 32), representing events that must eventually occur (an active `HotState` at the end of monitoring results in an error report). The monitor detects all illegal transitions, such as when a command to cancel preheating is dispatched in a state where the system is not preheating (Line 7); here the Boolean expression '`not self.PreheatState()`' denotes a query on the monitor memory for the non-existence of a `PreheatState` state. The states `PreheatState` and `WaitingForDriveCommandState` themselves define state local transition functions, which are applied to all events submitted to the monitor as long as the states are active. The result `pyc.ok` causes the executing state to be removed. Likewise `pyc.error` causes an error to be recorded.

A monitor can be viewed as an extended state machine where the state is a collection of variables, and transitions can contain statements that update these variables, including for example collecting statistics. An important argument for using a library written in an expressive programming language rather than a narrow domain-specific language for temporal logic is the expressive power this yields. In addition, states themselves can be parameterized with data, making the formalism very convenient. PyContract offers many other features, including chaining monitors, which we use to let one monitor M_1 process a stream of low-level events to produce a stream of high-level events that are monitored by another monitor M_2. Time is represented as just another kind of data (time stamps) in events, and therefore does not require extra machinery. Several kinds of states (beyond `HotState`) are offered, including non-hot states, conjunction, disjunction, negation, sequencing, and others. Monitors can be optimized using event key hashing. Finally, monitors can be composed to form a modular tree structure.

The library also supports generating state machine diagrams from monitors. Figure 17 visualizes the monitor in Figure 16. These diagrams can be used to confirm by inspection the correctness of the monitor implementation against the property that the user has in mind. In Figure 17, a yellow state represents a hot state, which must be left eventually before the end of monitoring. When running a test through the test generation framework, the results of the monitors are shown in a status report, as shown in Figure 18. The report shows two monitor violations, one of which is elaborated in detail (expanded when clicking on the error). The goal of the report is to allow users to quickly identify which monitors (requirements) were violated, and then inspect the trace of events that led to the violation.

Reactive Monitors

Beyond passive observation, monitors can issue messages to the SUT when specified temporal conditions hold (by adding code to the monitor as discussed above). This is typically done during testing to cause changes in the SUT's behavior at given points during its execution. For example, while it is possible to include a time parameter that is used to determine when to trigger a thermal fault, this is generally not advisable. Adding new parameters in general increases the computational complexity of the search space during test case generation, and a time parameter can take on a potentially infinite number of values. In practice, especially for system level autonomy, it is generally not necessary to cause a thermal fault at any possible time, but instead only after particular events such as before, during, or after state changes, or when goals are completed such as reaching a waypoint. A monitor can instead listen for the events from the system level autonomy that report when a waypoint is reached and then trigger a thermal fault.

8. Conclusions and Future Work

Space missions could benefit from autonomous operations. However, the control system in charge of understanding the

state of the environment, and making rational decisions must be designed and tested according to a methodology that provides an adequate level of assurance. The methodology cannot purely rely on testing in an operational environment simply because the environment may not be accessible, or may even be only partially known.

We have presented a model-based, requirement-driven approach to address this problem. We introduced a standard decomposition of an autonomous system and the definition of a requirement tree to generate software specifications. When followed, this process yields software specifications that are less ambiguous and more complete than traditional manual approaches. We have presented an automatic test generation and monitoring framework that uses a range of algorithms to maximize several coverage metrics that are essential to autonomy assurance.

Our future work in the area of early design verification and validation includes the use of formal methods. In the early phases of the engineering life-cycle, it is crucial to design a system that satisfies critical properties by design with high probability, and these properties are difficult to fully assess using testing only. In the area of the test generation and monitoring, future work will involve expanding upon the definition of coverage in Equation 1 to investigate additional and more efficient methods of maximizing coverage of the parameter, requirement, and behavior spaces. We will also investigate ways of determining which parameters should be included in the test generation to ensure adequate testing.

9. Acknowledgments

This project is part of a Strategic Research and Technology Development (SRTD) task led by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

References

[1] *NASA Systems Engineering Handbook*. SP-2016-6105 Rev2. National Aeronautics and Space Administration, 2016.

[2] *NASA System Safety Handbook Volume 2: System Safety Concepts, Guidelines, and Implementation Examples*. Tech. rep.

[3] *Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners*. Tech. rep. Dec. 2011.

[4] Steven L. Cornford et al. "NASA Quality Assurance in an MBSE World". In: *2020 Annual Reliability and Maintainability Symposium (RAMS)*. ISSN: 2577-0993. Jan. 2020, pp. 1–6. DOI: 10.1109/RAMS48030.2020.9153683.

[5] Shreya Parjan and Dan Gaines. ""In OBP We Trust": Verification and Validation of the M2020 On Board Planner Flight Software". In: *2024 IEEE Aerospace Conference*. ISSN: 1095-323X. Mar. 2024, pp. 1–11. DOI: 10.1109/AERO58975.2024.10521132.

[6] Aaron Stehura. "Mars 2020 Terrain Relative Navigation Verification & Validation". In: ().

[7] Gregory Villar et al. "Mars 2020 Entry, Descent, and Landing Verification and Validation". In: (2020).

[8] Issa A.D. Nesnas, Lorraine M. Fesq, and Richard A. Volpe. "Autonomy for Space Robots: Past, Present, and Future". In: *Current Robotics Reports* 2.3 (Sept. 2021), pp. 251–263. ISSN: 2662-4087. DOI: 10.1007/s43154-021-00057-2.

[9] NATO. *NATO GLOSSARY OF TERMS AND DEFINITIONS*. 2020.

[10] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. pearson, 2016.

[11] Aaron D. Ames et al. "Control Barrier Functions: Theory and Applications". In: *2019 18th European Control Conference (ECC)*. June 2019, pp. 3420–3431. DOI: 10.23919/ECC.2019.8796030.

[12] Alessandro Pinto. "Analysis and Design of Uncertain Cyber-Physical Systems". In: *Computation-Aware Algorithmic Design for Cyber-Physical Systems*. Springer, 2023, pp. 25–53.

[13] ISO/TC 22/SC 32. *ISO 21448:2022 Road vehicles — Safety of the intended functionality*. June 2022.

[14] Nancy Leveson. "STPA Handbook". In: ().

[15] *iso 26262-3: Road vehicles – Functional safety – Part 3: Concept phase*.

[16] *GODA: A Goal-Oriented Requirements Engineering Framework for Runtime Dependability Analysis*.

[17] James S. Albus et al. "4D/RCS Version 2.0: A Reference Model Architecture for Unmanned Vehicle Systems". In: *NIST* (Aug. 2002). Last Modified: 2021-10-12T11:10-04:00 Publisher: James S. Albus, Hui-Min Huang, Elena R. Messina, Karl Murphy, Maris Juberts, Alberto Lacaze, Stephen B. Balakirsky, Michael O. Shneier, Tsai H. Hong, Harry A. Scott, Frederick M. Proctor, William P. Shackleford, John L. Michaloski, Albert J. Wavering, Thomas Kramer, Nicholas Dagalakis, William G. Rippey, Keith A. Stouffer, Steven Legowik.

[18] Rodney A Brooks. "Intelligence without representation". In: *Artificial intelligence* 47.1-3 (1991). Publisher: Elsevier, pp. 139–159.

[19] Issa A Nesnas et al. "CLARAty: An architecture for reusable robotic software". In: *Unmanned ground vehicle technology V*. Vol. 5083. SPIE, 2003, pp. 253–264.

[20] Nikolai Matni, Aaron D. Ames, and John C. Doyle. *Towards a Theory of Control Architecture: A quantitative framework for layered multi-rate control*. arXiv:2401.15185 [cs, eess, math]. Jan. 2024. DOI: 10.48550/arXiv.2401.15185.

[21] Nikhil Vijay Naik, Alessandro Pinto, and Pierluigi Nuzzo. "Contract Embeddings for Layered Control Architectures". In: *ACM Trans. Embed. Comput. Syst.* (Aug. 2025). Just Accepted. ISSN: 1539-9087. DOI: 10.1145/3764587.

[22] Alessandro Pinto, Anthony Corso, and Edward Schmerling. "Leveraging Compositional Methods for Modeling and Verification of an Autonomous Taxi System". In: *2023 IEEE International Conference on Assured Autonomy (ICAA)*. IEEE, 2023, pp. 34–43.

[23] Manuel Mazo Jr et al. *A Contract Theory for Layered Control Architectures*. arXiv:2409.14902 [eess]. Sept. 2024. DOI: 10.48550/arXiv.2409.14902.

[24] Michel D. Ingham et al. "Onboard Planning and Execution of Mobility and Telecommunications for the Endurance Lunar Rover". In: *AIAA AVIATION FORUM AND ASCEND 2024*. AIAA Aviation Forum and ASCEND co-located Conference Proceedings. American Institute of Aeronautics and Astronautics, July 2024. DOI: 10.2514/6.2024-4889.

[25] James T Keane. "Endurance: Lunar South Pole–Aitken Basin Traverse and Sample Return Rover". In: ().

[26] Yorie Nakahira et al. “Diversity-enabled sweet spots in layered architectures and speed–accuracy trade-offs in sensorimotor control”. In: *Proceedings of the National Academy of Sciences* 118.22 (June 2021). Publisher: Proceedings of the National Academy of Sciences, e1916367118. DOI: 10.1073/pnas.1916367118.

[27] Albert Benveniste et al. “Contracts for system design”. In: *Foundations and Trends in Electronic Design Automation* 12.2-3 (2018). Publisher: Now Publishers, Inc., pp. 124–400.

[28] Inigo Xabier Incer Romeo. “The Algebra of Contracts”. PhD Thesis. UC Berkeley, 2022.

[29] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[30] Ilche Georgievski and Marco Aiello. *An Overview of Hierarchical Task Network Planning*. arXiv:1403.7426 [cs]. Mar. 2014. DOI: 10.48550/arXiv.1403.7426.

[31] K. J Åström. “Optimal control of Markov processes with incomplete state information”. In: *Journal of Mathematical Analysis and Applications* 10.1 (Feb. 1965), pp. 174–205. ISSN: 0022-247X. DOI: 10.1016/0022-247X(65)90154-X.

[32] Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers, Second Edition*. Google-Books-ID: l50DEAAAQBAJ. Princeton University Press, Feb. 2021. ISBN: 978-0-691-19398-4.

[33] Mica R. Endsley. “Toward a theory of situation awareness in dynamic systems”. In: *Human Factors: The Journal of Human Factors and Ergonomics Society* 37 (1995), pp. 32–64.

[34] M. Fox and D. Long. “PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains”. In: *Journal of Artificial Intelligence Research* 20 (Dec. 2003), pp. 61–124. ISSN: 1076-9757. DOI: 10.1613/jair.1129.

[35] Shufang Zhu and Giuseppe De Giacomo. “Synthesis of Maximally Permissive Strategies for LTLf Specifications”. In: vol. 3. ISSN: 1045-0823. July 2022, pp. 2783–2789. DOI: 10.24963/ijcai.2022/386.

[36] Klaus Drager et al. “Permissive Controller Synthesis for Probabilistic Systems”. In: *Logical Methods in Computer Science* Volume 11, Issue 2 (June 2015). Publisher: Episciences.org. ISSN: 1860-5974. DOI: 10.2168/LMCS-11(2:16)2015.

[37] Martina Troesch et al. “MEXEC: An Onboard Integrated Planning and Execution Approach for Spacecraft Commanding”. In: ().

[38] Shivkumar Shivaji, Natalia Lobakhina, and Lucas Cordeiro. “PyBMC – a Bounded Model Checker for Python”. Work in progress. 2025.

[39] *The ESBMC model checker*. https://github.com/esbmc/esbmc.

[40] *FME Industry website: Tool Support for Formal Methods*. https://www.fmeurope.org/industry/tool-support.

[41] Wenhao Ding et al. “A Survey on Safety-Critical Driving Scenario Generation—A Methodological Perspective”. In: *Trans. Intell. Transport. Sys.* 24.7 (July 2023), pp. 6971–6988. ISSN: 1524-9050. DOI: 10.1109/TITS.2023.3259322. URL: https://doi.org/10.1109/TITS.2023.3259322.

[42] Anthony Corso et al. “A Survey of Algorithms for Black-Box Safety Validation of Cyber-Physical Systems”. In: *Journal of Artificial Intelligence Research* 72 (Oct. 2021). ISSN: 1076-9757. DOI: 10.1613/jair.1.12716. URL: http://dx.doi.org/10.1613/jair.1.12716.

[43] Tommaso Dreossi et al. “VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems”. In: *31st International Conference on Computer Aided Verification (CAV)*. July 2019.

[44] Aman Sinha et al. “Rate-Informed Discovery via Bayesian Adaptive Multifidelity Sampling”. In: *8th Annual Conference on Robot Learning*. 2024.

[45] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. In: *CoRR* abs/1712.05889 (2017). arXiv: 1712.05889.

[46] Takuya Akiba et al. “Optuna: A Next-generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[47] Linbin Yu et al. “ACTS: A Combinatorial Test Generation Tool”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 370–375. DOI: 10.1109/ICST.2013.52.

[48] Martin Pelikan, David E. Goldberg, and Erick Cantú-Paz. “BOA: the Bayesian optimization algorithm”. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*. GECCO’99. Orlando, Florida: Morgan Kaufmann Publishers Inc., 1999, pp. 525–532. ISBN: 1558606114.

[49] Nikolaus Hansen, Sibylle D. Müller, and Petros Koumoutsakos. “Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES)”. In: *Evolutionary Computation* 11.1 (Mar. 2003), pp. 1–18. ISSN: 1063-6560. DOI: 10.1162/106365603321828970.

[50] Kevin Jamieson and Ameet Talwalkar. *Non-stochastic Best Arm Identification and Hyperparameter Optimization*. 2015. arXiv: 1502.07943 [cs.LG].

[51] Lisha Li et al. *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization*. 2018. arXiv: 1603.06560 [cs.LG].

[52] Hugo Araujo, Mohammad Reza Mousavi, and Mahsa Varshosaz. “Testing, Validation, and Verification of Robotic and Autonomous Systems: A Systematic Review”. In: *ACM Trans. Softw. Eng. Methodol.* 32.2 (Mar. 2023). ISSN: 1049-331X. DOI: 10.1145/3542945. URL: https://doi.org/10.1145/3542945.

[53] Wei Zheng et al. “ISTA: Automatic Test Case Generation and Optimization for Intelligent Systems based on Coverage Analysis”. In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2023, pp. 758–762. DOI: 10.1109/SANER56733.2023.00086.

[54] J. C. Gower. “A General Coefficient of Similarity and Some of Its Properties”. In: *Biometrics* 27.4 (1971), pp. 857–871. ISSN: 0006341X, 15410420.

[55] Dennis Dams, Klaus Havelund, and Sean Kauffman. “A Python Library for Trace Analysis”. In: *22nd International Conference on Runtime Verification (RV)*. Ed. by Thao Dang and Volker Stolz. Vol. 13498. LNCS. Springer International Publishing, 2022, pp. 264–273.

[56] *PyContract*. https://github.com/pyrv/pycontract.