Towards Architecture-wide Analysis, Verification, and Validation for Total System Stability During Goal-Seeking Space Robotics Operations

Catharine L. R. McGhan* and Yuh-Shyang Wang† and Richard M. Murray‡
California Institute of Technology, Pasadena, CA, 91125, USA.

Tiago Vaquero§ and Brian C. Williams¶
Massachusetts Institute of Technology, Cambridge, MA, 02139, USA.

Michele Colledanchise‖ and Petter Ögren**
KTH - Royal Institute of Technology, Stockholm, Sweden.

In this paper we discuss the beginnings of an attempt to define and analyze the stability of an entire modular robotic system architecture – one which includes a three-tier (3T) layer breakdown of capabilities, with symbolic, deterministic planning at the highest level. We approach the problem from the standpoint of a control theory outlook, and try to formalize the issues that result from trying to quantitatively characterize the overall performance of a well-defined system without a need for exhaustive testing. We start by discussing the concept of bounded-input bounded-output stability, giving examples where the technique might not be sufficient to guarantee what we term "total system stability" due to complications associated with the levels of abstraction between the modules and components that are being chained together in the architecture. We then go on to discuss necessary conditions that may fall out of this naturally as a result. We further try to better-define the input and output constraints needed to guarantee total system stability, using an assumption-guarantee-like contractual framework that sits alongside the architecture; the requirements then may have influence across multiple modules, in order to keep consistency. We also discuss how the structure of the architectural modules may help or hinder the process of capability characterization and performance analysis of each module and a given architecture configuration as a whole. We then discuss two overlapping methods that, combined, should allow us to analyze the effectiveness of the architecture, and help towards verification and validation of both the components and the system as a whole. Demonstrative examples are given using a specific architectural implementation called the Resilient Spacecraft Executive. In future work, we hope to define both necessary and sufficient conditions for total system stability across such a system architecture for robotics use.

I. Introduction

In the next decade, space exploration missions are being sent to destinations that are more remote and hazardous to answer more sophisticated science questions; as a result, in order to advance the knowledge frontier, the less-accessible targets need to be visited in more difficult, harsh, and inaccessible environments.

*Postdoctoral scholar, Department of Control and Dynamical Systems, 1200 E. California Blvd., Mail Code 305-16, Member.
†Graduate student, Department of Control and Dynamical Systems, 1200 E. California Blvd., Mail Code 305-16.
‡Professor, Department of Control and Dynamical Systems, 1200 E. California Blvd., Mail Code 107-81.
§Postdoctoral scholar, Department of Aeronautics and Astronautics, 32 Vassar Street, 32-224, Member. Joint appointment with Caltech.
¶Professor, Department of Aeronautics and Astronautics, 77 Massachusetts Avenue, 33-330, 32-227, Member.
‖Graduate student, Centre for Autonomous Systems, Teknikringen 14, Zip code 114 28, Room 615.
**Associate Professor, Centre for Autonomous Systems, Teknikringen 14, Zip code 114 28, Room 708.

This leads to many new challenges that will require a greater capability for onboard reasoning. We will need to be able to revise science objectives 'on the fly' as circumstances quickly evolve in these uncertain environments under short timescales; we will need those systems to be resilient to failure and able to gracefully degrade under harsh, long-term conditions. The Resilient Spacecraft Executive (RSE) design (shown in Figure 1) is an example of the sort of architecture that will enable these capabilities; it is meant to autonomously run onboard the spacecraft, making decisions in real-time when remote missions are being conducted that cannot include ground control in the loop from Earth, due to the timescales and delay involved.[1]

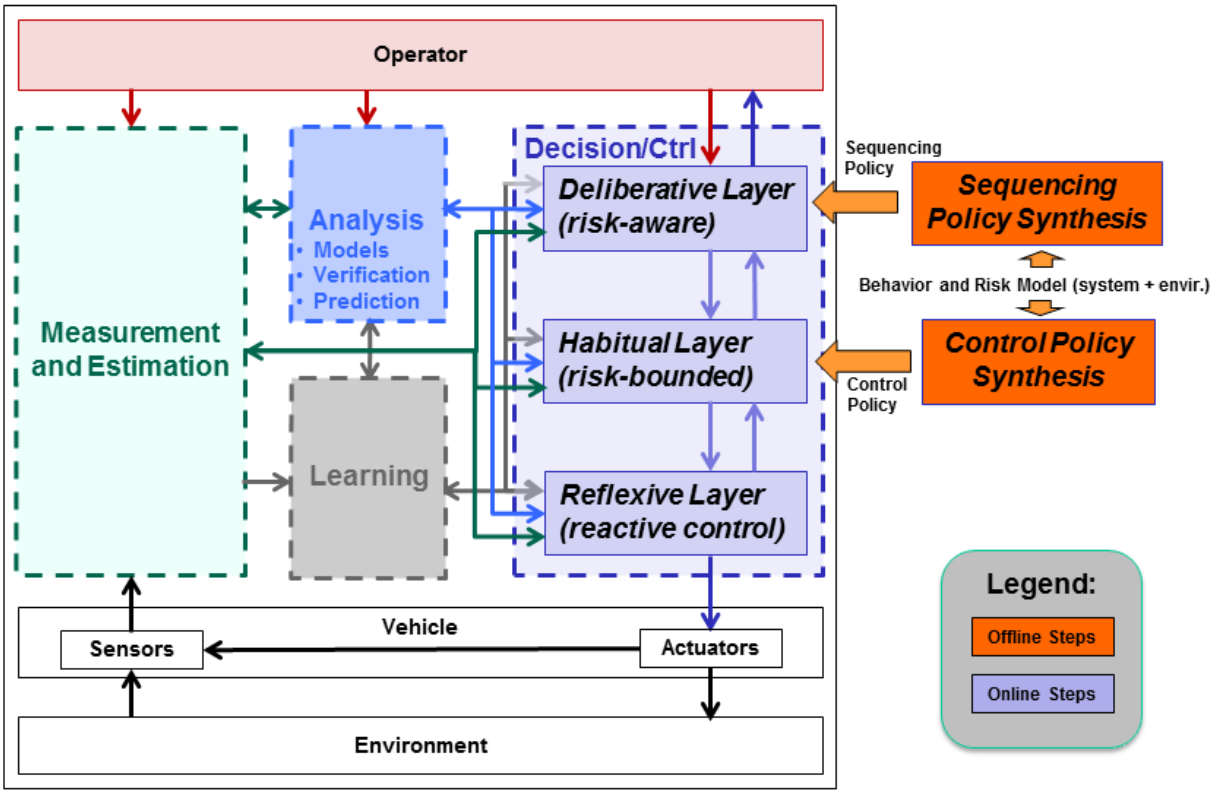


Figure 1: Resilient Spacecraft Executive Architecture.

And, while we wish to implement such a framework without unduly increasing system complexity, the complexity of these systems will necessarily increase to afford these new capabilities. As a result, we need to find new and better methods of guaranteeing that an acceptable baseline of mission performance is achieved with these new systems. Also, while environmental uncertainty may be a key limiting factor that requires us to extend our algorithms to handle such sources and circumstances that provoke it, it is also a problem when it comes to verification and validation of these new algorithms and systems. Preferably, we would wish to find ways to do so without exhaustive testing of the system, which simply cannot be done in all ways and combinations for all possible internal reconfigurations and all scenarios that might be encountered over the mission lifetime.

One way we intend to avoid an unnecessary increase in complexity is via the use of verifiable auto-coding software and correct-by-construction techniques that can synthesize policies and controllers according to the specifications given. However, these systems must be designed 'to-spec' in the first place. Some constraints and system requirements are imposed on the architectural modules and their relationship to each other due to the mission goals and the assumptions made about the operating environment. Others necessarily will need to be imposed due to the choice of algorithms and models used as part of the system architecture onboard the vehicle (e.g., to achieve certain goals within an allowable risk level, replanning in real-time may not be a feasible solution, but contingency planning might). Hard, physical constraints on performance also come from the robotic system itself, due to limits on power and computing resources, and the capabilities of the hardware components (e.g., maximum torque able to be applied via actuator, maximum allowable force that the vehicle structure can withstand). There are also constraints that can come from a mix of these, such

as a "reaction time" constraint, as the total time delay from sensor input to actuator output must be less than a given amount for system stability to even be possible – this would be dependent upon a given robotic system. An example of some robot-specific dependencies would be: the location of sensors and actuators, the sensor accuracy, level of uncertainty and sensor noise, the bounds on actuator performance, and the control algorithms used (e.g., PID versus PD versus MPC versus other methods). All of these various constraints and bounds on performance must be taken into account, in order to create consistent requirements for each architecture module, so that we can effectively analyze system performance and use formal methods to verify whether the operation of a specific architecture implementation will or won't achieve the operator-supplied mission goals.

In this paper, we describe what it means to have a 'stable architecture', which we term as having 'total system stability' (TSS), and give examples to show the complexity of the validation problem, in the context of the RSE architecture for a rover scenario example. We discuss how methods such as assumption-guarantee contracts may help us define and map overarching constraints from a qualitative representation to a more useful and directly implementable quantitative one. We also detail an example internal structure for such architectural modules, and why we believe that using this structure should help the analysis and verification and validation (V&V) process, both in terms of capability characterization and performance analysis of each module and for a given architecture configuration as a whole. We then go into detail on two overlapping methods that should help in our goal of analysis and V&V of the architecture, towards increasing confidence in these more complex systems. We show demonstrative examples of applying these methods to our test RSE implementation, and discuss directions of future work, towards defining sufficient conditions for total system stability, as well as methods for automatic verification of each module, such as the autocoding of the components from the specifications.

II. RSE Architecture

To ground our discussion, 'total system stability' will be discussed in the context of the RSE architecture, which was created for resilient risk-aware goal-achievement (and risk-bounded planning and execution) in the presence of both environmental uncertainty, and sensor and actuator degradation and failures.

We have discussed in prior work[1] some examples of (limited) resilient behavior on spacecraft, such as the Remote Agent Experiment flown on Deep Space One,[2,3] the autonomous navigation capability on Deep Impact,[4,5] and the Cassini spacecraft's onboard orbit insertion calculations.[6] We have also discussed layered autonomy architectures, like Remote Agent, CLARAty[7,8] and the Autonomous Sciencecraft capability on Earth Observing One.[9] To restate, the key distinctions and innovations in the RSE framework (shown in Figure 1) include, as originally given in:[1]

(i) The development and use of formal architectural analysis to perform tradeoffs and inform the appropriate allocation of capabilities to the deliberative, habitual and reflexive layers. This will result in systems with flexibility to adapt to their uncertain environments and potential mission changes. This is in contrast to the informal allocation of capabilities to layers in current architectures, which results in brittle architectures with properties that are inappropriately tuned to the mission context (e.g., favoring responsiveness over flexibility, even for mission scenarios without strict time-criticality requirements).

(ii) The architecture's leveraging of sequencing and control policies that are "correct by construction" in both the deliberative and habitual layers. The use of model-based policy synthesis will address the current challenge of assuring correctness of the system behavior in the face of growing complexity.

(iii) The use of onboard deliberative reasoning, which will enable the system to manage a space of possible executions that is far too large to be completely covered by design-time control policies, and light-time delays that preclude effective ground-based deliberation and planning for many future mission scenarios.

(iv) The architecture's emphasis on risk-awareness, which is critical to managing the unprecedented amount of uncertainty in the environments to be explored in future missions. Such uncertainty introduces significant risk and precludes any guarantees of correct behavior, even though we are employing formally correct-by-construction policies. Endowing our architecture with the ability to explicitly assess risk and make decisions based on risk fills this resilience gap.

These innovative features makes it possible for our RSE architecture to allow autonomous operation that is resilient enough to support space exploration in uncertain and high-risk environments, and overall more ambitious science collection capabilities. However, this requires a modular architecture with a breakdown of capabilities over levels of decomposition.

A. Algorithms

The deliberative layer performs *risk-aware plan execution*. A risk-aware plan executive takes as an input a set of high-level goals (i.e., a plan), makes risk-aware decisions, and outputs subgoals that are executed by the habitual layer. Risk-aware plan execution is distinct from conventional planning in two ways:

1. A risk-aware plan executive adapts its decisions to the acceptable level of risk specified by users.
2. A risk-aware plan executive performs *risk allocation* among the subgoals it generates.[10]

The first essentially addresses the trade-off between risk and utility, and the second scales the first capability when there is more than one subgoal. Note that one can think of a robot as having a limited amount of risk (resource) that it can use to achieve a goal, like fuel or energy; allocating risk optimally across the subgoals then maximizes overall utility.[10]

Various algorithms and plan executives have been developed by MIT's Model-based Embedded and Robotic Systems (MERS) Group, which has been a pioneer in risk-aware plan execution. The iterative risk allocation (IRA) algorithm[10] provides the optimal risk allocation capability for a wide range of problems, and was the basis for *p-Sulu*.[11] The algorithm is built upon chance-constrained model predictive control (CCMPC) methods[12–14] and works on a continuous state space; two of its current applications are vehicle path planning[10,15,16] and building control.[17] More specifically, p-Sulu takes as input a plan representation called chance-constrained qualitative state plan (CCQSP)[18] and outputs an optimal sequence of actions as a schedule. These algorithms are part of what the MERS Group calls their Enterprise system.

Other Enterprise system components include probabilistic risk-aware activity planners and schedulers, such as continuous-p-Sulu, ptBurton, and pKirk. These planners can deal with temporal uncertainty, uncertainty in action-completion, and uncertainty in action-outcome.

TuLiP is an implementation of one of a set of new approaches that has been created within the last decade for the specification, design, and verification of embedded control systems (see[19] for an overview and additional references). These approaches make use of models of the dynamics of the system, descriptions of the external environment, and formal specifications to either verify that a given design satisfies the specification, or synthesize a behavioral policy or controller that satisfies the specification, as summarized in Figure 2.

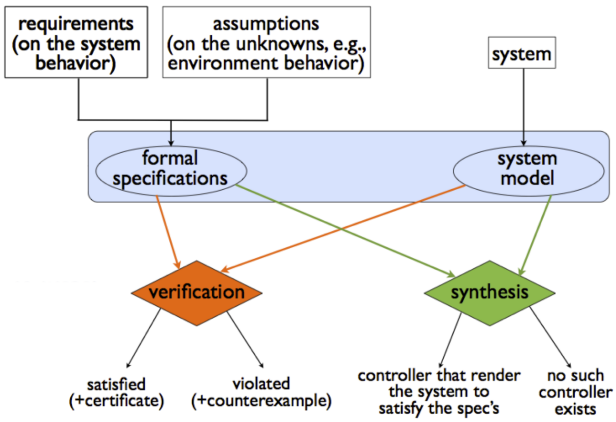


Figure 2: Verification and synthesis framework.

More specifically, TuLiP is a correct-by-construction synthesizers for the creation of behavioral policies and controllers; it makes use of a dynamic model of the system, descriptions of the external environment, and formal specifications to either verify that a given design satisfies the specification or synthesize a controller that satisfies the specification, as summarized in Figure 3. TuLiP can be used for symbolic activity planning

to find a satisficing (not optimal) plan,[20] and also in conjunction with the Enterprise system components as a plan verification tool for the creation and evaluation of contingency plans at planning time. TuLiP can also be used to create hybrid control policies at the lower layers.[21]

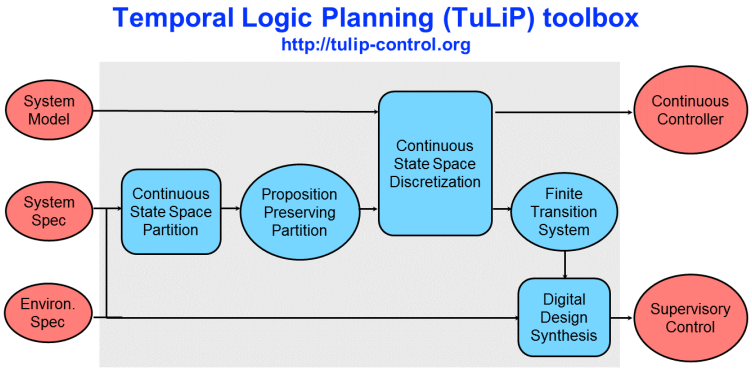


Figure 3: TuLiP software code structure flowchart. Blue boxes are TuLiP capabilities (functions) and blue circles are data representations, while red circles are inputs (left) and outputs (right). Note that the system model could be input as LTL specifications or as a finite transition system; system and environment specifications are linear temporal logic formulas.

The habitual layer utilizes *risk-bounded hybrid control* techniques to perform 'rote' behaviors that will achieve the deliberative layer goals, while also satisfying the safety and/or performance constraints specified by the deliberative layer. In the rover scenario case, one of the common behaviors performed at this level is trajectory planning. The reflexive layer uses existing low-level control techniques, such as PID control.

For more information on the RSE architecture implementation, including algorithms that have been integrated and made available in the codeset, see.[22]

B. RSE Architecture Module Structure

Before we can discuss the structure of each component module in RSE, we need to discuss the outcomes we are trying to achieve here, and the capabilities we are trying to support with RSE (e.g., the analysis and verification of the entire architecture, graceful degradation and resilient operations).

In the big picture, what we are trying to do is to create an architecture that allows for the following:[23]

1. Resilience Quantified in Measurable Terms
   - Project cost, Mission risk, Quality(/opportunity) of science return
2. Resilient Architecture Tradeoffs
   - Passive vs. active resilience, Reflexive vs. deliberative, Resilience/robustness vs. efficiency/cost effectiveness, Complexity vs. verifiability, Hierarchical vs. flat architecture
3. Resilient Architecture Principles
   - Options (diversity or redundancy of), Modularity and interfaces (constraints that de-constrain), Ability to choose (between options), Characterization of uncertainties ('Resilient to what?')

To support these resilience needs, we must attempt to create an implementation that easily allows for:

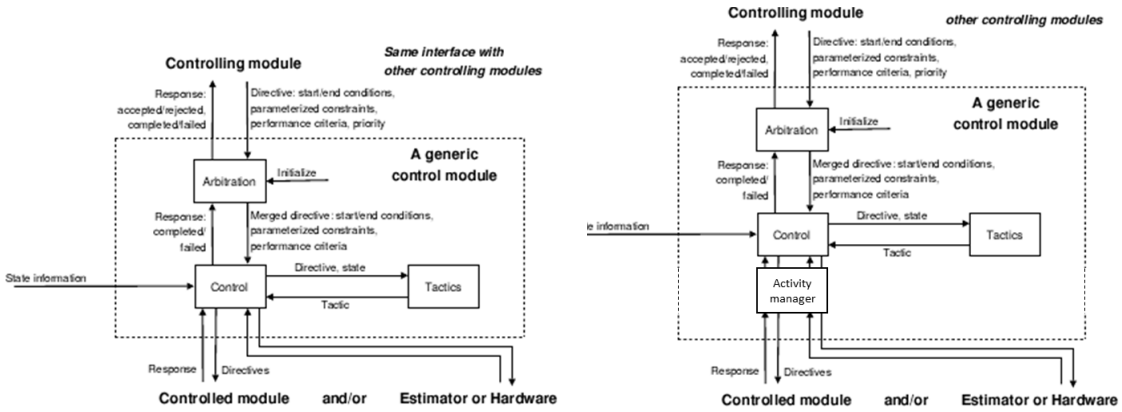
(1) analysis and V&V of both the components and the architecture as a whole

(2) multiple types of configurations, differing levels of abstraction

Note that item (1) points towards the need for an architectural analysis framework that can be used to evaluate an RSE architecture specification and the system-level design as a whole. This idea of across-architecture analysis leads to the idea of total system stability (discussed in section III), and a need for theory to describe and analyze such layered abstractions (discussed in section IV). Items (1) and (2) both

imply a need for common interfaces and 'plug-n-play'-type algorithms or modules, for interchangeability and apples-to-apples comparisons.

Also note that the structure of the architectural modules may help or hinder the process of capability characterization and performance analysis of each module and a given architecture configuration as a whole. Thus, in order to better support both of the above functionalities, for a fair assessment and comparison between one RSE architectural implementation and another, we need (1) our implementation to be modular, and (2) to be able to formally specify every aspect of the systems, techniques, and methods that we wish to use, as well as their interconnections and the robot procedures being used. In this effort, we have chosen to use the Canonical Software Architecture (CSA) format,[24,25] in order to support the decision/control layer decomposition at the levels of abstraction we choose, the separation of responsibility for degradations and failures, and the segregation of functionality, while still maintaining the necessary communication and contingency management between components in the architecture. CSA also supports our need to explicitly and formally specify the robot procedures and interconnections (rather than implicitly encode them into the architectural structure). Figure 4a shows an example CSA module.



(a) Original CSA module.[24,25] (b) 'Version 2' with activity manager.

Figure 4: Canonical Software Architecture generic control modules, comparison between versions.

Note that CSA builds off of the state analysis framework developed at JPL,[26–29] and that another reason we use CSA is because allowing only one source of state knowledge to each module prevents the modules from getting out of sync and helps to disallow inconsistency in state knowledge.

Each generic CSA module can be broken down into an Arbitration, Control, and Tactics component:[24,25]

- Arbitration manages the overall behavior of the module by issuing a merged directive computed from all the received directives to Control, and reports goal status back to the issuing module.
- Control computes the output directives to other module(s) based on Arbitration's merged directive, responses received from other modules, and state information received; it also reports failure and completeness of a merged directive to Arbitration.
- Tactics generates a control tactic or a contiguous series of control tactics for Control to use.

In the RSE implementation, each of these module components follows an explicitly-defined policy that chooses the component's internal action (algorithms) and can change the module's or the component's internal status, based upon the content of the various inputs to the component and its current internal state. For instance, the Arbitration component can include a state machine that determines whether a goal and constraints are accepted or rejected, and handles status messages and requests between the modules; if a goal and constraints are accepted it passes the information along to the Control component to be processed further.

In order to better support architectural analysis and multiple types of configurations, we extend the original CSA module specification to include an activity manager (see Figure 4b). The extended CSA 'version 2' specification still includes (a) message bundling and constraints on acceptable I/O interface, (b) explicit segregation of arbitration, control, tactics inner modules, (c) similar internal variable passing as

defined by CSA, and (d) the ability to start/stop/reset each module externally during runtime. The new 'activity manager' component below 'control' is similar to the 'arbitration' component above, but more of an algorithm registry. Enacting this promotes ease of communication with separate algorithm modules and lower levels, by finishing what the 'arbitration' component started, separating (all) the external communication procedures (e.g., "state machine"(s)) from the 'control' component.

This separation of the activity manager and control aspects allows us to (similarly) separate and modularize the response of the algorithms from the more-complex layer response. Instead of having to include multiple algorithms inside a single deliberative layer module, multiple different algorithms can then 'register' with the activity manager. One reason we may want to have separate algorithm modules able to be called would be to allow for parallel computation (possibly even across different CPU cores), for consensus or first-viable-response responses. With several algorithm modules 'registered' to a single layer, each algorithm can be called upon (through the activity manager) by the control component as-needed, for short- or long-term calculations, in sequence or in parallel. Figure 5 shows an example of a more complex 'stacking' of CSA modules and capabilities.

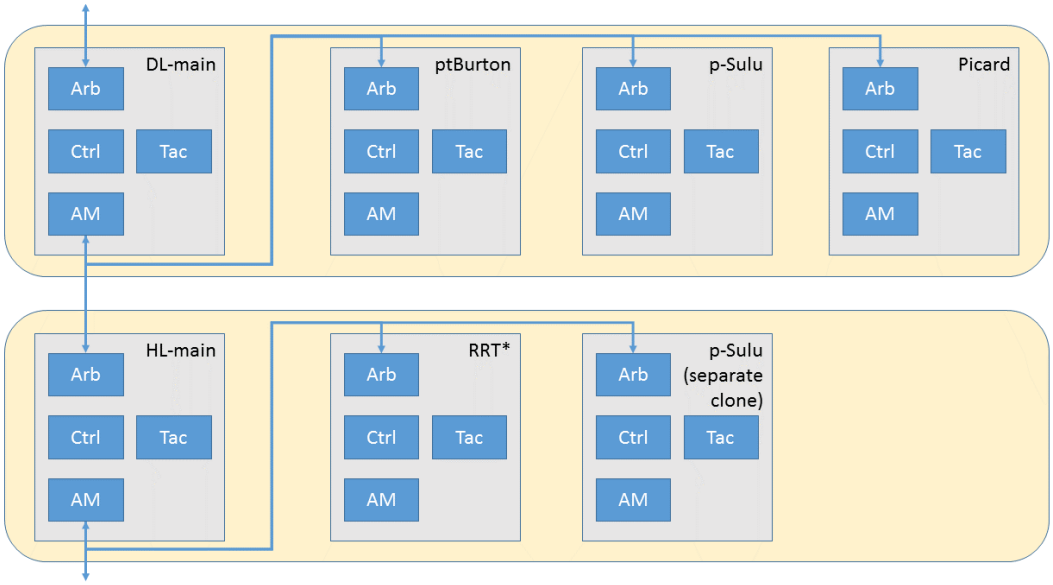


Figure 5: An example of within-layer stacking of CSA modules.

Another reason we might wish to separate the activity manager comms and policy functionality from the control component is to allow for ease of use of the RSE architecture for single-domain experts. Someone who is an expert at a particular algorithm (its strengths and weaknesses and capabilities, who knows whether it can complete in hard real-time and whether it can give guarantees of correctness and/or completeness in certain domains) should not have to understand state machines and the RSE layer policies to try and offer up their algorithm for use, or to test their algorithm within a pre-existing architecture implementation. Sharing the API at the arbitration component for their new algorithm module should be enough to register that algorithm for use, if it uses a common specification; the control component of a layer module can then send computation requirements (problem formulation, constraints, etc.) to the layer's activity manager component, and that activity manager could select or ping each registered algorithm module to determine what capability set (and associated algorithm) would best solve the given problem. If a solution cannot be found within the constraints given (as reported by the algorithm module), then the activity manager can report this back to the control component, and the control component's policy can handle the 'no solution found' case as we would expect it to (either internally, or failing up through the arbitration component). Thus, more complex reactions and policy responses can be encoded for each layer, largely independent of the main control component policy and the specific algorithms chosen – rather, policy responses can be determined according to algorithm class, control mode, or tactic, instead.

Note that this decomposition and modularity also makes both general testing/debugging and V&V far

easier. By having explicit state machines(/policies) for the internal layer operations, we make both single-module V&V easier (we know how to analyze a policy), and multi-module V&V possible (interactions between state machines, environment). It also becomes possible to use correct-by-construction techniques to synthesize these policies (e.g., we can lay out formal requirements that each algorithm must satisfy).

In the next section, we discuss a specific type of architectural analysis that is a first-step towards being able to make formal guarantees about goal-completion in the presence of uncertainty. While the ability to perform V&V across an architecture implementation is important, it does not necessarily guarantee that the goals given to that architecture will be able to be completed. Being able to determine whether an architecture performs according to a set of specifications as-given, or whether its policies are consistent, is different than determining what specifications are necessary (and/or sufficient) for resilient operations to occur in a given scenario, under a particular set of assumptions.

III. Total System Stability

In this section, we discuss what it means to have a 'stable architecture', and how to guarantee that an architecture is "stable". We approach this problem from the standpoint of a control theory outlook, and below we try to formalize the issues that result from trying to quantitatively characterize the overall performance of a well-defined system without a need for exhaustive testing.

For this purpose, we define the term "total system stability" (TSS) as 'goal-tracking' behavior from the highest level of abstraction (dealing with mission goals and constraints) through the lowest level of controller output. In other words, as controller stability is defined by whether a given controller is able to converge to a reference trajectory input, total system stability is defined by whether the entire architecture is able to 'converge' to a set of given goals – to complete those goals. Thus, "total system stability" can also be described as stability of the entire architecture. As an example, for a controller, we ask the question: can we track / converge upon the reference trajectory? For an architecture, we would ask the question: can we track / converge upon the system/mission goals? "Converging on the goals" can be thought of as accomplishing the set of goals as-given. Thus, if a subset of the goals is not accomplished, then we are not tracking well, and if none of the goals are accomplished, we have diverged (not convergence).

A. Reasoning

With a multi-layer architecture, replanning and failing-up from lower levels to higher levels is fine as long as the goals are (eventually) accomplished within the constraints given (e.g., time, fuel use, risk level), assuming goal completion is possible under the goals and constraints given. (The architecture should only fail-up if it must.) This can impose further constraints on the system that are derived from the robots physical model (e.g., sensor and actuator specs) and the environment (e.g., uncertainty in sensing, type of obstacle and classification of risk); some of these constraints include the amount of acceptable end-to-end time delay in the system (e.g., absolute time from sensor reading to higher-level computation to control output), the allowable max/min speed for the robot (e.g, to avoid catastrophic damage from impact with unavoidable obstacles or obstacles that are sensed too late), and so forth.

From a control theory standpoint, one might naively assume that total system stability could be obtained by applying the idea of bounded-input bounded-output (BIBO) stability to the system. Being able to prove BIBO stability for each CSA module could then guarantee suboptimal but acceptable performance for the overall system. However, there are several problems with this (not the least of which is: where do the input and output constraints come from, and what should they be?).

First, in order to have this work, the output from each module must lie within the input bounds for the module it is giving its output to. If the constraints do not match for BIBO stability, there will be issues. The problem here is matching up those constraints, primarily because of the differences in levels of abstraction between the modules. It is very hard (and possibly overconstraining) to make sure that each module has full coverage of the information from its input source, because each higher-level module is operating at that higher level of abstraction for a reason – namely, to reduce the size of the state space to make the problem size tractable. Modules operating at a lower level of abstraction may not be able to handle the explosion of the state space that might result.

The second problem with the idea of BIBO stability in this instance is that determining what the bounds (i.e., input constraints) need to be for any given module, in order to prove BIBO stability for that module,

is highly nontrivial for a resilient architecture. For example, proving that a trajectory planning module can always find a path (when one exists) that will get a rover from a specific location A to a specific location B within a polygon-area of a given map, in less than a given time T, might be possible for a deterministic environment. However, when there is a probability that a 'pop-up obstacle' might be sensed during traversal that must be avoided, one which was not on the original map, the input bounds – the assumptions made – have been violated. Thus, that proven trajectory, with guarantee of arrival within time T, no longer applies. (In other words: when there is uncertainty in the environment, this further complicates matters.) Similar issues result when a fault or degradation occurs (in either the sensors or actuators).

However, we don't want this to be a show-stopper. If there is only a 5% chance of that rover encountering and sensing an obstacle of radius R that was not on our original map, we would want some guarantee still that we could still accomplish at least the majority of our high-level goals, even if we might have to punt on the current traversal task and not visit location B until later (...or not at all, in order to still complete as many goals as possible). This becomes even more interesting when our high-level goals are more fluid things, like "obtain at least X pieces of such-as-such type of data", or "collect Y reward level of science data, weighted by priority and amount and usefulness", or "do not exceed K amount of overall risk during rover operations". Recomputing the schedule of tasks for the day to still meet these demands might still be feasible (as opposed to a set of tasks sent down by mission control that must be completed for a 'success'ful outcome over that planned time period).

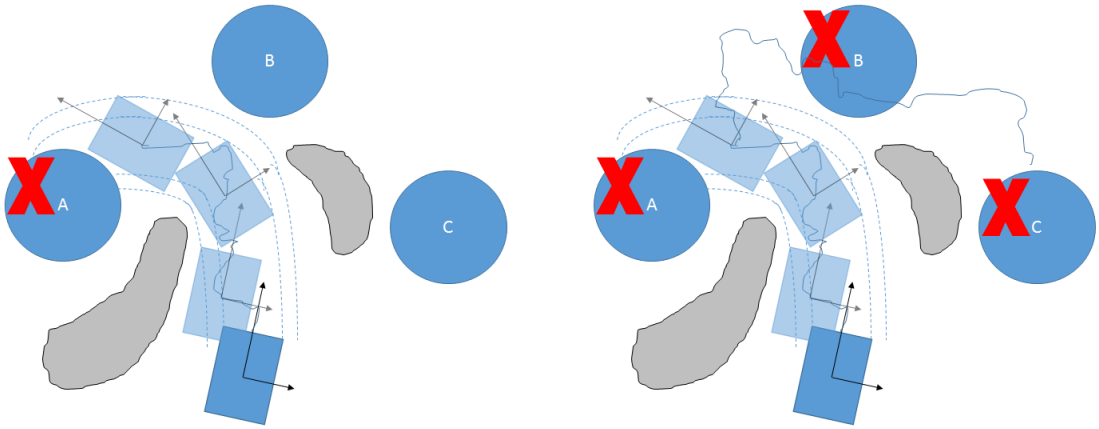
This gets back to the reason why we have multiple layers and multiple levels of abstraction in the first place: no single module or layer should be responsible for everything. We want to allow for the replanning and rescheduling of tasks when problems arise. Again, replanning and failing-up from lower levels to higher levels is fine as long as the goals are (eventually) accomplished within the constraints given.

However, allowing this results in further complexity and additional issues. If the levels of abstraction between layers aren't chosen well for a particular pairing of problem instantiation (e.g., rover scenario) and architecture implementation, problems can develop. For instance, when the constraints sent down between layers are too tight, thrashing between the two layers can occur (unnecessary, internal delay); alternately, when those bounds are too loose, there may be times when fail-up needs to occur but doesnt happen – when higher-level replanning was required, but didn't happen, or didn't happen in time.

B. Examples of Constraint Mismatch

Because of the need to make some sort of guarantee on performance (probabilistic or absolute) even in the face of uncertainty, BIBO stability for each module/component may not be enough to ensure total system stability without consistency across the entire architecture. As discussed in later sections of this paper, we do have some methods of ensuring consistency and/or goal convergence between the habitual and reflexive layers, and between the deliberative and habitual layers. And, the performance bounds (restrictions, constraints, and/or requirements) given between each decision/control layer, along with the transformed more-concrete behaviors for goal completion, may seem reasonable as defined between each set of layers. However, while it is essential to have consistency between each pair of layers, the layer-pairs of these constraints may not guarantee overall stability of the system on their own. This is due to complications associated with the levels of abstraction between the modules and components that are being chained together in the architecture. Ensuring consistency across all three layers is another matter entirely! The complexity of the validation problem cannot be easily circumscribed in a multi-layer (¿2 layers) system, simply by sticking to BIBO stability, as shown below in the context of a Mars rover scenario. For instance, when a simple 'fail-up when a constraint bound is violated' policy is used for each layer, there are circumstances where the constraints chosen may seem reasonable at first, but are too 'loose' or too 'tight' to prompt a 'fail-up' to occur at the appropriate time. And, as said before, when the constraints do not match, total system stability cannot necessarily be ensured.

In this first example, shown in Figures 6a and 6b, setting bounds that are too tight can cause an inability for the system to track its goals. For instance, if the deliberative layer has a heuristic of the lower level(s), the deliberative layer sends the habitual layer a bound on time of completion, and the habitual layer sends the reflexive layer a tight bound on the trajectory envelope (for obstacle avoidance), a case of unexpected slippage can cause internal thrashing between the habitual and reflexive layers. The reflexive layer would try to track the trajectory, run outside the bounds of the trajectory envelope due to the high rate of slippage, and have to fail up to the habitual layer. The habitual layer would spend time replanning, compute a slightly-different trajectory with bounds that are just as tight (the constraints on safe obstacle avoidance



(a) First location (A) cannot be reached within given time bound T, due to too-tight combination of high slippage rate, trajectory bound, and non-zero duration computational thrashing between the layers.

(b) Locations B and C can also be chosen that cannot be reached within the time bound T, for a given slippage rate, trajectory bound, and non-zero trajectory computation time.

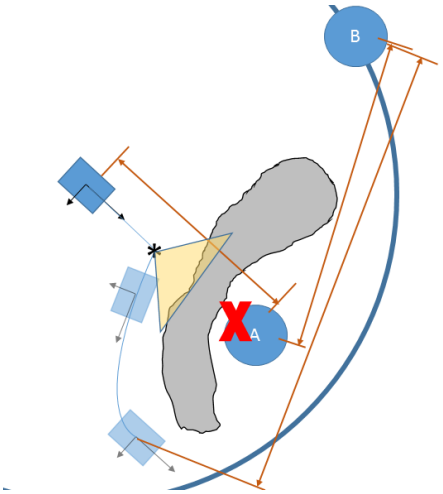
Figure 6: Rover example: bounds too tight. Grey blobs are obstacles, blue circles with letter designations are goal locations. Red 'X's show locations that were not able to be reached within the given time bound.

not having changed in the meantime), and send down the new trajectory. The reflexive layer would only be able to follow the new trajectory for a short distance... and the whole process repeats until the time bound on arriving at its destination (A) is about to be violated. The habitual layer would fail up at this point, the deliberative layer would have to cross that first destination point (A) off the goal list as unable to be completed, and would move on to trying to reach the next destination point (B).

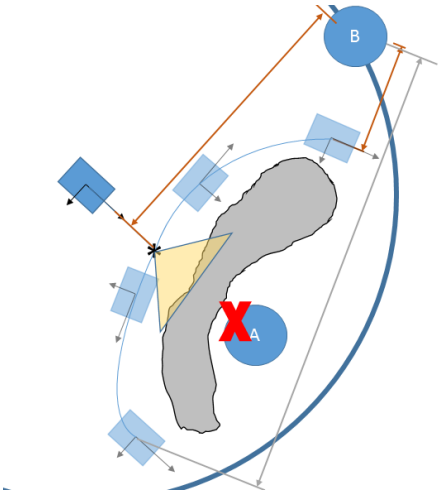
The real issue here is that the habitual layer was given too much responsibility on the correction of slippage but not enough leeway on time. In this case, the deliberative layer slippage model doesnt match the trajectory bound (being sent to the reflexive layer from the habitual layer) or the time of completion (being sent to the habitual layer). A pathological case (set of destination points) can be chosen for a given scenario (level of slippage and trajectory bound, versus time of completion computed from deliberative layer heuristics) that would always have the destination point not being reached and having to be stricken off the goal list, for every destination point on the list.

In the second example, shown in Figure 7a, setting bounds that are too loose can also lead to an inability for the system to track its goals. For instance, if the deliberative layer has a heuristic of the lower level(s), the deliberative layer sends a large bound on the time of completion, and the habitual layer sees part of an unexpected obstacle that was not on the original map that the deliberative layer used for calculation, there can be problems if the habitual layer computes that it can get around the obstacle with (what would otherwise be thought to be) an acceptable level of probability of successful arrival within the time bounds. As shown in Figure 7a, if the rover moves around the obstacle counter-clockwise, and (eventually) can't get to location A in time (because the obstacle is too large/long to traverse around even within the given very loose time bound), it will fail up to the deliberative layer, which will cross that destination point (A) off the goal list as unable to be completed, and would move on to trying to reach the next destination point (B). However, by this point, the rover would be much farther away from destination (B) than before, and (a pathologically-chosen destination point B can always be chosen for any given time horizon that it) can no longer be reached!

The real issue here is that the habitual level was given too much responsibility on pop-up obstacle avoidance, with no sense of the next goal (reaching the destination point). Unfortunately, this cannot simply be solved by giving the habitual layer the next goal, or the next, or the next – one can always pick a longer horizon than the current timeline that will force this pathological behavior to happen. Simply having the habitual layer fail-up every time that an unexpected obstacle is encountered is also no solution – this can lead to similar thrashing as discussed above in the habitual-reflexive layer example of too-tight bounds. The bigger issue here is the division of responsibility between the layers. The deliberative layer's responsibility is

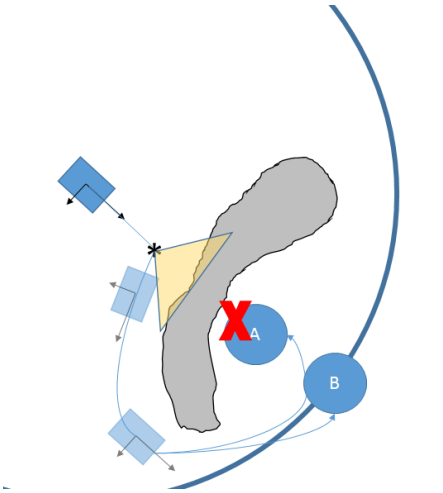


(a) First location (A) cannot be reached within given time bound T1. Due to rover's uninformed choice to traverse counter-clockwise around the obstacle in trying to reach location (A), the rover has ended up far enough away from the second location (B) that it can no longer reach it within the original time bound T2.

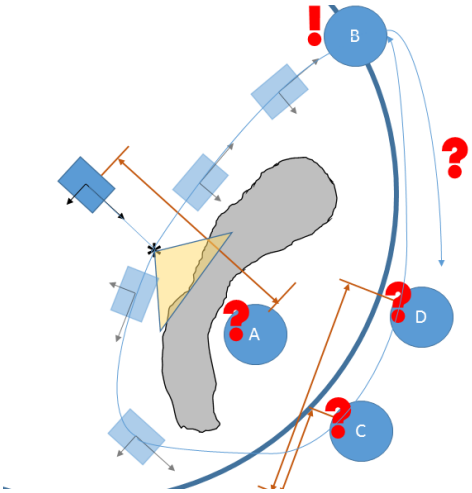


(b) First location (A) cannot be reached within given time bound T1. Here, with an informed choice by the rover to traverse clockwise around the obstacle, the rover has not ended up farther away from location (B) than it would have been had it successfully reached location (A), and can still reach the second location (B) within the original time bound T2.

Figure 7: Rover example: bounds too loose. Grey blobs are obstacles, blue circles with letter designations are goal locations. Red 'X's show locations that were not able to be reached within the given time bound.



(a) First location (A) cannot be reached within given time bound T1. Here, the rover having moved around the obstacle in a counter-clockwise manner is all right, because even though the obstacle was not accounted for in the higher-level plan, in this case moving towards the first location (A) was also moving the rover closer to the second location (B).



(b) First location (A) is expected to be reached within given time bound T1, as computed upon first sensing of the obstacle. The habitual layer does not know the priorities of reaching the locations (A, B, C, D) relative to each other. The best handling of this case might be to fail-up and request replanning occur, given this new information.

Figure 8: Rover example: bounds too loose. Grey blobs are obstacles, blue circles with letter designations are goal locations. Red 'X's show locations that were not able to be reached within the given time bound.

the discrete goal timeline and goal tradeoffs/priorities. The habitual layer should only have to think about the limited timeline that it is given; the constraints that are given to the layer should allow it to fail-up as and when it is necessary for the deliberative layer to replan for circumstances that must (and do) affect the global timeline (in most cases adversely).

Thus, the focus shifts to determining what constraints – what metrics – must be defined that can be used to guarantee that a layer will always fail up when it needs to, in time for the higher-level layer to replan and handle the circumstances accordingly. Figure 7b showcases one possible 'constraint' that could be used for this purpose (a necessary, though likely not sufficient condition, for guaranteeing fail-up at the appropriate time and thus total system stability). When the rover first senses the unexpected obstacle, it had an unremarked-upon choice of which direction to attempt to traverse around the obstacle: clockwise, or counter-clockwise. If the habitual layer only has knowledge of destination A for whatever reason, it would need to fail-up to the deliberative layer. Alternately, if the next destination point (B) had been passed down to the habitual layer, then the rover could have computed that it would be 'safer' to move clockwise around the obstacle, so that in the event that the rover did not reach destination (A) in time, it would at least not be farther away from destination point (B) than it was at the start of its traversal. (Note, however, that this is dependent upon the location of the next destination point. Figure 8a shows an example where moving counter-clockwise around the obstacle would get the rover closer to the next destination point (B).) This 'simple fix' is not, however, without its pitfalls. Allowing the habitual layer to make these decisions on its own – whether to fail up or not – discounts the strengths of the deliberative layer. Figure 8b shows an example of a larger plan with additional destinations to be reached, where the rover can exact different scientific measurements. When the priority of reaching those destinations comes into play, it quickly becomes clear that the decision of the rover's direction of movement (or a reassignment of goals) is actually something that should by right be handled by the deliberative layer. For instance, if reaching destination (B) is essential, and reaching destination (A) was merely added to the plan due to the rover's perceived proximity on the way to destination (B) (without having had knowledge of the blocking obstacle), and destinations (D) and (C) added to the end in case a nominal traversal occurred and the destinations were able to be reached afterwards, the deliberative layer might choose to reassign the rover to reaching destination (B) immediately upon sensing the obstacle, and drop destination (A) from the list entirely because of this. However, if all four destinations had equal value in their science return, then dropping destination (B) from the goal list and continuing to traverse around the obstacle to destinations (C) and (D) would be more efficient, instead.

C. Recommendations

There are no simple solutions, here. However, we do have some recommendations on necessary (not sufficient) conditions to help mitigate the issues shown in these two examples, without unnecessarily trying to muddy the distinction between the types of computation that could or should be done between the layers (or the assignment of responsibilities for unexpected events and/or hardware functionality degradations between them). (One is parallel computation, where the higher levels sense and replan without requiring a fail-up, and can send 'repaired' plan information down to lower levels as necessary. However, the same metrics to sense these cases to determine whether intervention is in order are what would also be used as the constraints to determine an explicit fail-up from a lower layer itself.) Our first recommendation is that each constraint sent down is calculated according to the cost associated with changing the plan (to help prevent thrashing between the layers). Our second recommendation is to look for possible reevaluation points or circumstances (like the new obstacle sensed that was not on the map used by the deliberative layer in developing the original higher-level plan). The evaluation done at the lower level (e.g., habitual layer) should not just be a simple 'accept'/'reject' of whether it is expected to be able to be accomplished or not; instead, this should be an evolving real-time status and prediction of outcome. The lower (habitual) level should continually check to see if it has violated or will violate 'the assumption that the layer can (still) complete the current goal'. (The closest formal example to this is a contract guarantee, an assumption that the module will do what it said it was going to do, what it thinks it can do.) Our third recommendation is that, along with the second recommendation, 'point(s) of no return' should be identified – points in the timeline when the action must be completely committed to. An example metric for this is for the percentage of completion to be known to be above "b" by time "tx", and the likelihood of completion to also be above a threshold "c2"; if this constraint was not met, then one would move to a contingency plan (precomputed by a higher, e.g. deliberative, layer) or fail-up. This metric can also be used as a more general constraint to determine when to send updates to the higher level layer(s), or when to prompt a replan – for instance, if the likelihood of

completion drops below "c2" at any point, or if the likelihood of completion drops more than a "range c3" between a set number of timesteps/sensor readings "nx".

Overall, we are trying to make sure that unexpected events fail-up properly (e.g., only when they need to in order to have the highest rate of goal achievement possible), and this is a question of bounds and what our constraints sent down to the lower layers needs to be. Also, if enough fail-ups occur in a short period of time, we may expect to start to see thrashing between the layers. So to avoid this, we might also want to use the frequency of fail-up occurence as a measure of when we might need to recalculate our models (e.g., model fitness), and/or update the parameters used for the computation of the confidence of completion within bounds (e.g., the progress along a path coupled with the predicted future progress).

D. Ongoing Work

This is only the start of attempting to deal with this problem. Future work on this will involve continuing to determine necessary and sufficient conditions for consistency (including V&V on the internals of each CSA module). As briefly touched upon above, identifying reevaluation (or decision) points is one thing that needs to be done. However, this does not negate the need for the constraints being used to be consistent across all levels of abstraction, all modules and layers. (In other words, all the "BIBO" constraints across all components must match!)

We also need to formally describe what assumptions are being made, and the guarantees that are needed, which in turn drive the constraints that must be defined. Thus, a cataloging of assumptions (and physical constraints) needs to be made at every level of the architecture (and likely for each use case, though the methods for performing this determination should remain somewhat the same). This will likely impact the types of layer/module constraints, and the actual quantitative numbers that are calculated. This catalog will live in a "parallel plane" alongside the architecture that keeps track of the global (and module-specific) assumptions being made, since it will necessarily have a multi-module, multiple-layer impact (not just one above-below), and the requirements that are defined by this should be traceable to each constraint that is defined by this process. Having a separate, explicit catalog of these will allow us to (1) better track whether anything that is correct-by-construction will fail (as incomplete models with incorrect assumptions lead to a case of garbage in, garbage out), and (2) determine when our models no longer hold and when learning is needed. (Note that in graceful degradation and failure cases, we will expect to see changes to the assumptions; thus, we will want to use fault trees, or a similar method, to enumerate the causes and consequences of failures, and this will determine the modes of operation needed for the system implementation.) We plan to use assumption-guarantee contracts to help us define and map these overarching constraints from the more qualitative representation in the catalog to a more useful and directly implementable quantitative one. Assumptions should lead to guarantees and constraints on the system. Guarantees should be clear (accept/reject, status) and violation (or prediction of a violation with high probability) should immediately prompt a fail-up.

At a meta-level, it should be noted that as part and parcel of determining these "BIBO" constraints, we are effectively determining the functional requirements for each module / layer in the architecture. This can help define which algorithms are useful; it can help inform what set (or sets) of algorithms should be picked by an operator for a successful (stable) architecture implementation, for a given robot and problem scenario. (Note that this should fall out naturally from the abstraction breakdown; the robot capabilities; and the mission(/scenario), including such details as the level of uncertainty in the occurrence of certain events and the acceptable risk bound, which further define the overall assumptions and constraints necessary.) From this, we would then want to create a database of algorithm capabilities, characteristics, and evaluation metrics, in order to better match what properties and performance are needed by a module in the architecture implementation with a particular algorithm or set of algorithms. Some things that we believe it would be helpful to characterize would be: the general set of functional capabilities that describe it (e.g., the sort of API information given to the activity manager); the performance & accuracy of the algorithm as impacted by the calculation-time/CPU-cycles allotted to it and the size of the problem space; and common "hooks" that can be exploited for higher efficiency runs (e.g., restrictions on the breadth and depth of search, calculation time, and/or iterations performed).

Finally, it should be noted that the formal determination and mapping of constraint-dependencies is not always straightforward. For instance, to have a resilient architecture, some of the metrics we look at are:

1. "Project cost" (such as money, human time, and resources)

2. "Mission risk" (the ability to perform all mission goals)

3. "Quality(/opportunity) of science return" (some preferred tradeoff of "a lot of data" versus the "usefulness of that data")

From this we can derive additional requirements for resilient operations, e.g.:

- the sensors and actuators necessary to complete an action must be functional during these times (functional components)
- the graceful degradation of components and/or intelligent acquisition and relay of data (/ intelligent planning of sensor and actuator use, including the transmission windows)

These then lead to further requirements:

- to limit / avoid the possibility of breakage when possible (e.g., obstacle avoidance, smart on/off of components)
- the allocation of satellite time, relay, CPU time/compression, data rates, data bus, memory storage for storing and relaying data

Note that, for a rover scenario, we as humans might think of obstacle avoidance as one of the first problems to occur to us. However, obstacle avoidance is really only a consequence of the need to avoid damage to the rover. And, when a rover mission is at the end of its lifecycle, if the rover is sat down in a very interesting location, where it is taking all the final science data that is needed to complete its mission, the ability to traverse is no longer of overriding importance. Similarly, if a rover can only take X number of samples using a manipulator arm, and has done so, then a failure of one of the joints of the arm that would not allow it to pull itself into a stowed position might not be considered more than a minor inconvenience. If the arm has a science instrument on the outstretched end effector and can still take other measurements, the event of that actuator joint dying would no longer be considered the stop-stopping critical and crippling failure event that it would have been if it had happened much earlier in the mission, when there were still sample slots to be filled. The severity of a degradation or failure is dependent on not just the event, but also when it occurs in the mission timeline and what goal-dependencies still must be met.

IV. Methods for Architecture Analysis

In this section we discuss two overlapping methods that should help in our goal of analysis and V&V of the architecture, towards increasing confidence in these more complex systems.

The first method uses behavior trees and encompasses the deliberative layer and the habitual layer, trying to map the physical space into (possibly overlapping) zones of discrete states, and try to identify which zones need to be explored for total system stability.[30] The second method uses robust control theory and encompasses the habitual and reflexive layers, trying to quantify the effects of bounded uncertainty on total system stability.

A. Method 1: Behavior Trees, DL-HL

Behavior Trees (BTs) are a graphical mathematical model for reactive fault tolerant execution of tasks. They were first introduced in the computer gaming industry[31,32] to describe the task execution of non-player-characters, meeting design requirement as modularity, flexibility and reusability.

Their popularity lies in their particular structure and execution; in particular, they have been shown to generalize both AND-OR-trees,[33] the Subsumption architecture, the Sequential composition,[34] Decision Trees and Teleo-reactive Paradigm[35] creating a growing attention in academia.[36–43]

1. Background

Here, we give a brief overview of BTs, and refer to[36] for a more detailed description.

A BT is defined as a directed tree where nodes are grouped into root; control flow nodes; and execution nodes, using the usual definition of *parent* and *child* for each connected nodes. The root node has no parents and only one child, a control flow node has one parent and at least one child, and a execution node has

no children and one parent (i.e. it is a leaves node of the tree). Graphically, the children of a control flow node are ordered from its bottom left to its bottom right, as in Figures 10a-10b. A BT starts its execution from its root node, which sends *ticks* to its child. (A tick is a signal that enables the execution of a child.) When a node in a BT receives a tick, its execution starts and it returns to its parent a status *running* if its execution has not finished yet, *success* if its execution is accomplished, or *failure* otherwise.

We now describe the execution of the the nodes aforementioned using the functional model proposed in.[38]

2. The Functional Model of BTs

Definition 1 (Behavior Tree (BT)[38]). *A BT is a three-tuple*

$$\mathcal{T}_i = \{f_i, r_i, \Delta t\}, \tag{1}$$

where $i \in \mathbb{N}$ is the index of the tree, $f_i : \mathbb{R}^n \to \mathbb{R}^n$ is the right hand side of an ordinary difference equation, Δt is a time step and $r_i : \mathbb{R}^n \to \{\mathcal{R}, \mathcal{S}, \mathcal{F}\}$ is the return status, that can be equal to either Running *($\mathcal{R}$),* Success *($\mathcal{S}$), or* Failure *($\mathcal{F}$).*

The return status r_i will be used when recursively combining BTs, as explained below.

Definition 2 (Execution model of a BT[38]). *The execution of a BT $\mathcal{T}_i$ is a standard ordinary difference equation*

$$\begin{aligned} x_{k+t}(t_{k+1}) &= f_i(x_k(t_k)), & (2)\\ t_{k+1} &= t_k + \Delta t. & (3) \end{aligned}$$

Definition 3 (BT regions[38]). *The three regions $R_i, S_i, F_i \subset \mathbb{R}^n$ of a BT $\mathcal{T}_i$ are defined as follows*

$$\begin{aligned} R_i &= \{x : r_i(x) = \mathcal{R}\} & (4)\\ S_i &= \{x : r_i(x) = \mathcal{S}\} & (5)\\ F_i &= \{x : r_i(x) = \mathcal{F}\} & (6) \end{aligned}$$

and denoted Running region (R_i), Success region (S_i) and Failure region (F_i).

(a) Graphical representation of an action node.

(b) Graphical representation of a condition node.

Figure 9: Graphical representation of the execution nodes.

Definition 4 (Condition). *A Condition is a BT $\mathcal{T}_i$ with $R_i = \emptyset$.*

The condition node checks if a condition is satisfied or not. The return status is success or failure accordingly and it is never running. The condition node is represented in Fig. 9b.

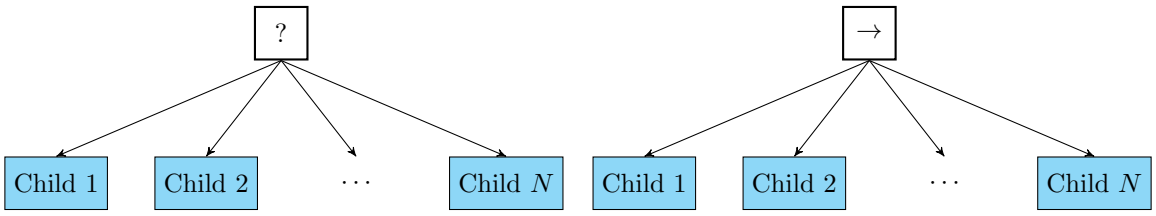
BTs that satisfy Definition 1 directly, without calling other subtrees, see below, is called Actions.

Definition 5 (Action). *An Action is a BT $\mathcal{T}_i$ that has no subtrees.*

When an action node starts its execution, then it returns success if the action is completed and failure if the action cannot be completed. Otherwise it returns running. The action node is represented in Fig. 9a

Definition 6 (Sequence compositions of BTs[38]). *Two or more BTs can be composed into a more complex BT using a Sequence operator,*

$$\mathcal{T}_0 = \mathit{Sequence}(\mathcal{T}_1, \mathcal{T}_2).$$



(a) Graphical representation of a fallback node with N children.

(b) Graphical representation of a sequence node with N children.

Figure 10: Graphical representation of the control flow nodes.

Then r_0, f_0 are defined as follows

$$\begin{aligned} &\textit{If } x_k \in S_1 && && (7)\\ &\quad r_0(x_k) &&= r_2(x_k) && (8)\\ &\quad f_0(x_k) &&= f_2(x_k) && (9)\\ &\quad \textit{else} && && \\ &\quad r_0(x_k) &&= r_1(x_k) && (10)\\ &\quad f_0(x_k) &&= f_1(x_k). && (11) \end{aligned}$$

$\mathcal{T}_1$ and $\mathcal{T}_2$ are called children of $\mathcal{T}_0$. Note that when executing $\mathcal{T}_0$ it keeps executing its first child $\mathcal{T}_1$ as long as it returns Running or Failure. The second child is executed only when the first returns Success, and $\mathcal{T}_0$ returns Success only when all children have succeeded, hence the name Sequence. For notational convenience, we write

$$\text{Sequence}(\mathcal{T}_1, \text{Sequence}(\mathcal{T}_2, \mathcal{T}_3)) = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3), \tag{12}$$

and similarly for arbitrarily long compositions.

When the execution of a fallback node starts (i.e. the node receives a tick from its parent), then the node's children are executed in succession from left to right, until a child returning success or running is found. Then, this message is returned to the parent of the fallback. It returns failure only when all the children return a status failure. The purpose of the fallback node is to robustly carry out a task that can be performed using several different approaches (e.g. a motion tracking task can be made using either a 3D camera or a 2D camera) by trying each of them in succession until one succeeds. The graphical representation of a fallback node is a box with a "?", as in Fig. 10a.
A finite number of BTs $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N$ can be composed into a more complex BT using the fallback composition: $\mathcal{T}_0 = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N)$.

Definition 7 (Fallback compositions of BTs[38])**.** *Two or more BTs can be composed into a more complex BT using a Fallback operator,*

$$\mathcal{T}_0 = \textit{Fallback}(\mathcal{T}_1, \mathcal{T}_2).$$

Then r_0, f_0 are defined as follows

$$\begin{aligned} &\textit{If } x_k \in F_1 && && (13)\\ &\quad r_0(x_k) &&= r_2(x_k) && (14)\\ &\quad f_0(x_k) &&= f_2(x_k) && (15)\\ &\quad \textit{else} && && \\ &\quad r_0(x_k) &&= r_1(x_k) && (16)\\ &\quad f_0(x_k) &&= f_1(x_k). && (17) \end{aligned}$$

$\mathcal{T}_1$ and $\mathcal{T}_2$ are called children of $\mathcal{T}_0$. Note that when executing $\mathcal{T}_0$ it keeps executing its first child $\mathcal{T}_1$ as long as it returns Running or Success. The second child is executed only when the first returns Failure, and

$\mathcal{T}_0$ returns Success when one children has succeeded, hence the name Fallback. For notational convenience, we write

$$\text{Fallback}(\mathcal{T}_1, \text{Fallback}(\mathcal{T}_2, \mathcal{T}_3)) = \text{Fallback}(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3), \tag{18}$$

and similarly for arbitrarily long compositions.

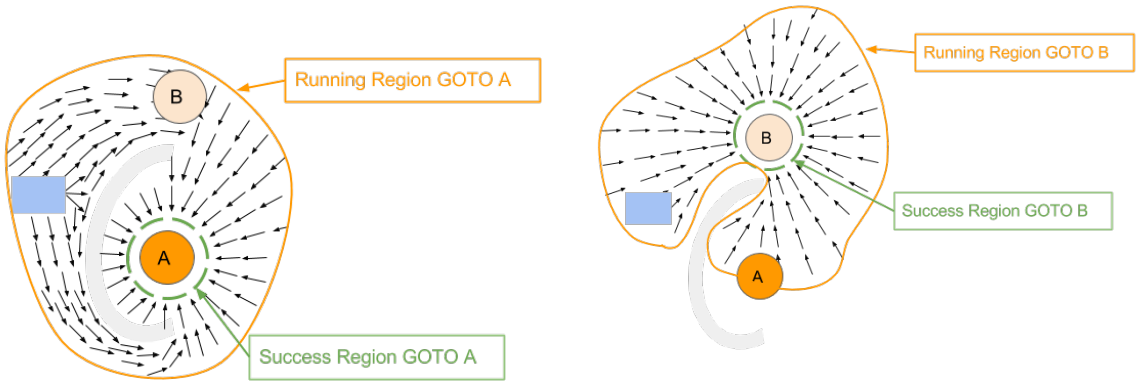
When the execution of a sequence node starts, then the node's children are executed in succession from left to right, returning to its parent a status failure (running) as soon as the a child that returns failure (running) is found. It returns success only when all the children return success. The purpose of the sequence node is to carry out the tasks that are defined by a strict sequence of sub-tasks, in which all have to succeed (e.g. a mobile robot that has to move to a region "A" and then to a region "B"). The graphical representation of a sequence node is a box with a "$\rightarrow$", as in Fig. 10b.
A finite number of BTs $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N$ can be composed into a more complex BT using the sequence composition: $\mathcal{T}_0 = \text{Sequence}(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_N)$.

Definition 8 (Root)**.** *The root node is the node that generates ticks. It is graphically represented by a box labeled with "$\varnothing$".*

3. Proposed Approach – Constraint passing DL-HL

To guarantee the *goal tracking behavior* from HL to DL we need to guarantee that the execution of one task does not impede the execution of another task. In a BT framework this is easily described by having the success region of a tree representing a task being inside the running region of the tree representing the next task. The running region of a task τ_i is defined as the points in the state space from which the system can reach the goal satisfying the given constraint. The success region of a task τ_i is defined as the points in the state space in which the goal is considered reached.

As stressed earlier, we account for fault at any time. Whenever a fault occurs, the Success and Running regions are updated according to the new constraints. The Running and Success regions for each task are computed at the HL level (only the HL algorithms know the possible points in the state space), whereas the order of these tasks are computed at the DL level. To guarantee the *goal tracking behavior* from HL to DL, the DL needs to constrain the success region of a task in order to ensure that after a task has succeeded, the next task is able to succeed from that point. Moreover, whenever a success region of a task is not inside the running region of the next task, the task is aborted.



(a) Regions for GOTO A.

(b) Regions for GOTO B.

Figure 11: Running regions (solid orange) and success regions (dashed green) of the subtrees GOTO A and GOTO B.

Example 1. *Imagine the system has to reach two waypoints A and then B as depicted in Fig. 11. Consider a waypoint that can nominally be reached if the system in within* 1m *from it. Each waypoint must be reached with some given time constraint. The constraint is translated into a set of points from which the system can reach the waypoint in time. As it can be seen in Figure 11 if the HL decides to compute a path that ends at the bottom part of Waypoint A, then from there the system will not be able to reach the waypoint B in*

Algorithm 1: main loop

```
1  init ← True
2  for i ∈ [1, m] do
3      send_constraints_to_HL(τ_i)
4  do
5      if check_faults_from_HL() or init then
6          init ← False
7          for i ∈ [1, m] do
8              R_i ← get_running_region_of(τ_i)
9              S_i ← get_success_region_of(τ_i)
10         for i ∈ [1, m] do
11             S_i ← S_i ∩ R_{i+1}
12             set_success_region_of(τ_i, S_i)
13             if not ( get_is_task_τ_i_aborted() ) and
14             not ( get_is_task_τ_i_aborted() ) and S_i ⊄ R_{i+1} then
15                 set_is_task_τ_i_aborted(TRUE)
16     r ←tick(T)
17 while r = running
```

time. Hence the success region of GOTO A needs to be restricted to be inside the running region of GOTO B. When computing the path to Waypoint A, the HL does not know that after reaching waypoint A it will be requested to compute a path to waypoint B, hence the restriction of the success region must be done on the DL level and sent down to HL.

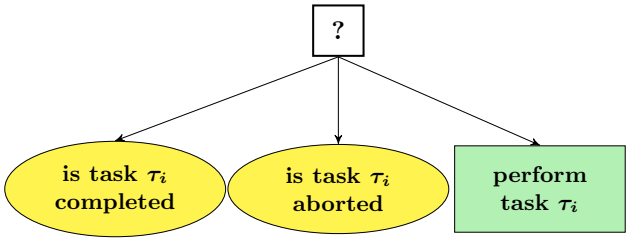


Figure 12: Proposed tree.

Algorithm 1 implements the procedure, which is formally described below:

Definition 9. *$\mathcal{T}_i$ is the tree representing a task τ_i in form of Fig. 12. Where: the condition node* is τ_i completed *returns success if the task is completed; the condition node* is τ_i completed *returns success if the task is aborted from the DL; and* perform τ_i *executes the task (sends the task to be performed down to HL);*

Definition 10. *$P = \{0, 1\} \times \{0, 1\} \times R^n$ is the state space in which $\mathcal{T}_i$ evolves. Where n is the cardinality of the state space.*

Definition 11. *$x_k = [x^c, x^a, x] \in P$ is the state of $\mathcal{T}_i$ (see Eq. (2)). Where $x_i^c = 1$ if and only if τ_i is completed; $x_i^a = 1$ if and only if τ_i is aborted; and x is the state variable of the system.*

Lemma 1. *Let $m \in \mathbb{N}$ be the number of tasks, n the cardinality of the state space, and $\mathcal{T}_i$ be a BT in form of Fig. 12, the tree $\mathcal{T} = sequence(\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m)$ can track a goal only if (necessary, not sufficient) each action* perform task τ_i *is stable and either one of the following holds: $x_i^c = 1$; $x_i^a = 1$ or $x \in R_i \implies x \in S_{i-1}$ where S_{i-1} is the success region of $\mathcal{T}_{i-1}$ and R_i is the running region of $\mathcal{T}_i$. By definition $S_0 = \mathbb{R}^n$.*

Proof. For each task τ_i one of three states can occur: τ_i is completed, τ_i is aborted or the system is performing τ_i. If τ_i was either completed or aborted one of the following holds: $x_i^c = 1$ or $x_i^a = 1$. If

the system is performing τ_i, the system state variable is in R_i it will be able to reach the goal only from S_{i-1} hence $x \in R_i \implies x \in S_{i-1}$. If $i = 1$ holds (i.e. the system is performing the first task), then $x \in R_1 \implies x \in S_0 = \mathbb{R}^n$ holds by definition. □

Algorithm 1 implements the aforementioned approach.

B. Method 2: Layering Architectural Analysis Framework, HL-RL

As stated above, this method uses robust control theory and encompasses the habitual and reflexive layers, trying to quantify the effects of bounded uncertainty on total system stability.

Layered control architectures have emerged in diverse application fields, including autonomous system, Internet,[44] process control,[45] smart grid,[46] biological systems,[47] and large-scale distributed systems.[48] Traditionally, these layered architectures are designed based on extensive experience and testing. This approach no longer holds for future engineering systems due to the rapidly increasing complexity of these systems. The *layering as optimization decomposition* framework proposed in[44] has proven to be a powerful mathematical theory in the design and analysis of network architectures. The basic idea is to formulate the system design problem as an optimization problem, and use a distributed algorithm to decompose the global optimization problem into simpler optimization subproblems in layers. Each subproblem corresponds to a functional module in the layered architecture, and the interfaces among layers are treated as variables to coordinate the subproblems. The theory has recently been extended to incorporate system dynamics and transient behavior.[48]

In this paper, we apply the layering as optimization decomposition framework to the design of a layered architecture in a robotics application. Different from the previous works,[44,48] which primarily focus on the efficiency and optimality of the architecture, we emphasize the analysis of the *resilience* of the layered architecture for RSE. Specifically, the layering as optimization decomposition framework focuses on a top-down optimization decomposition technique for functionality allocation, followed by proofs of optimality and convergence. We additionally incorporate uncertainty into the layered architecture, and propose a bottom-up approach to translate the bounds on the uncertainty set from lower level to higher level to ensure total system stability.

1. *Analysis*

We begin with the analysis of a two-layer hierarchical control architecture shown in Figure 13. The higher level controller is responsible for more sophisticated tasks (e.g., planning and scheduling), while the lower level controller is responsible for simpler tasks (e.g., tracking and regulation) but in a faster timescale. The aim of this is to analyze the total system stability of a hierarchical control architecture, and offer some guidelines for the design of a resilient layered architecture.

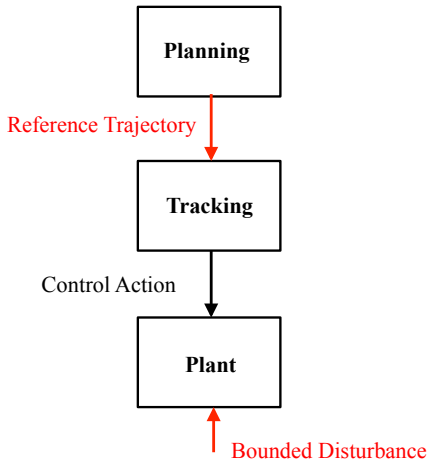


Figure 13: A Two-Layer Open Loop Hierarchical Control Architecture

Specializing our discussion to the rover traversal problem, the planning layer corresponds to the habitual layer in the RSE architecture, while the tracking layer corresponds to the reflexive layer. The overall goal is to find an obstacle-free path from source to destination, subject to the dynamics of the rover, hardware constraints, environmental uncertainty, and time constraints. This problem is highly complicated and cannot be solved at once. A tractable solution is to decompose the problem into two simpler parts: a planning problem, and a tracking problem. For the planning problem, we find an obstacle-free path while ignoring the detailed dynamics of the system. The reference path is then sent to the tracking layer in the lower level, where we design the control action to track the reference trajectory. This design methodology uses a top-down approach to decompose the task into multiple layers, as shown in Figure 13.

The layered control architecture in Figure 13 does not necessary guarantee total system stability because the controllers at different layers are designed based on models with different levels of abstraction. Specifically, we design the path planner to generate an obstacle-free path for the tracker, in the hope that the tracker can follow the reference trajectory nearly perfectly without hitting any obstacle. In practice, due to environmental uncertainty and the physical constraints of the rover, there are always some tracking errors that make the actual path deviate from the reference trajectory. If the trajectory deviation is large, the rover may hit an obstacle during its traversal. A common way to avoid this problem is to impose safety margins at the planning stage, which allows certain amount of trajectory deviation during actual implementation. Our goal is to propose a quantitative way to specify the safety margin at each layer to guarantee total system stability.

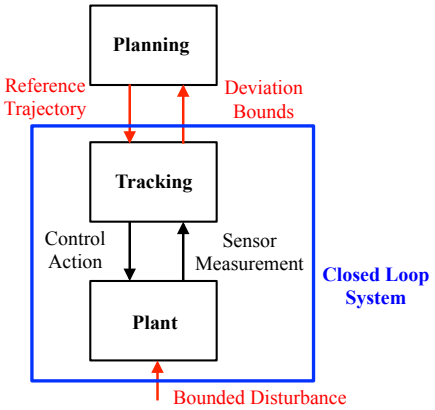


Figure 14: A Two-Layer Closed Loop Hierarchical Control Architecture

Consider the layering architecture shown in Figure 14. Here we provide a feedback signal between layers during both the synthesis and the implementation stage. Assume that the rover is subject to some bounded external disturbance. The role of the tracking layer is to use the sensor measurement to adjust fast timescale feedback control action to mitigate the effect of external disturbance on rover traversal. This part of the architecture can be designed using conventional feedback controller synthesis. The innovative part is that we further translate the bounds on the external disturbance to the bounds on trajectory deviation, and report the bounds to the planning layer in the design stage. The planning layer then checks whether the trajectory deviation is within its safety margin. If the maximum trajectory deviation for the given bounded disturbance is smaller than the safety margin, then we have a guarantee that the rover will not hit an obstacle during its actual traversal as long as the actual external disturbance is bounded in a pre-specified set. This provides an assumption-guarantee-like framework for verification and validation of total system stability. From Figure 14, we can also consider the tracking layer as an abstraction, or a virtualization of the rover dynamics and the hardware constraints to the planning layer. Specifically, the tracking layer translates the bounds on the external disturbance to the bounds on trajectory deviation. In this way, the planning layer does not need to consider the detailed dynamics of the system, which make the planning task simpler and faster.

The two-layer architectural framework in Figure 14 can be extended to multi-layer control architectures as well. For a multi-layer control problems, we first perform a top-down decomposition to allocate the functionalities into different modules in different layers. Then, we perform a bottom-up analysis to translate the lower level uncertainty set to a more abstract, higher level uncertainty set to ensure the total system stability of the layered architecture.

2. Design

In the previous subsection, we discuss the analysis of total system stability for a given controller. Here, we discuss the design of the controller in the layered architecture.

In Figure 14, we note that the combination of the tracking layer and the plant forms a *closed loop feedback system*. This closed loop system takes the reference trajectory and the external disturbance as inputs, and generates the deviation bounds as output to the planning layer. Suppose that the external disturbance has bounded magnitude (bounded ℓ_∞ norm), and we want to bound the magnitude of the trajectory deviation (bounded ℓ_∞ norm). The $\ell_\infty - \ell_\infty$ induced norm is known as the $\mathcal{L}_1$ norm[49] of the system. Therefore, if we want to minimize the magnitude of trajectory deviation for a given bounded external disturbance, then we solve an $\mathcal{L}_1$ optimal control problem to design the controller for the tracking layer. More generally, the work in[50] proposed a controller design framework that can incorporate constraints imposed on the closed loop system. The idea is to first design the desired closed loop behavior of the feedback system, then reconstruct the controller in the feedback loop. Therefore, the planner can first specify the constraints and objectives imposed on the closed loop system shown in Figure 14. We can then apply the method in[50] to synthesize the optimal controller for the tracking layer that satisfies the design specification given by the planning layer.

V. Preliminary Results of Applying Analysis Techniques

For the RSE architecture implementation, we adopted the Robot Operating System (ROS) messaging system (and ROSbridge) as the backbone for internal communications between components.[51,52] The publisher-subscriber model is fairly robust, and there exist a wide range of robots and simulated robots that have pre-existing interfaces to the software package, which will allow us to easily test our architecture across a wide range of use cases. Messaging support has been tested between the RSE and two simulation environments: a mid-fidelity rover simulation using the OSRF-developed gazebo software and a low-fidelity simulation using the Pioneer 3-DX rover.[53] The gazebo software allows for a wide range of robots to be tested using the main RSE software backbone.

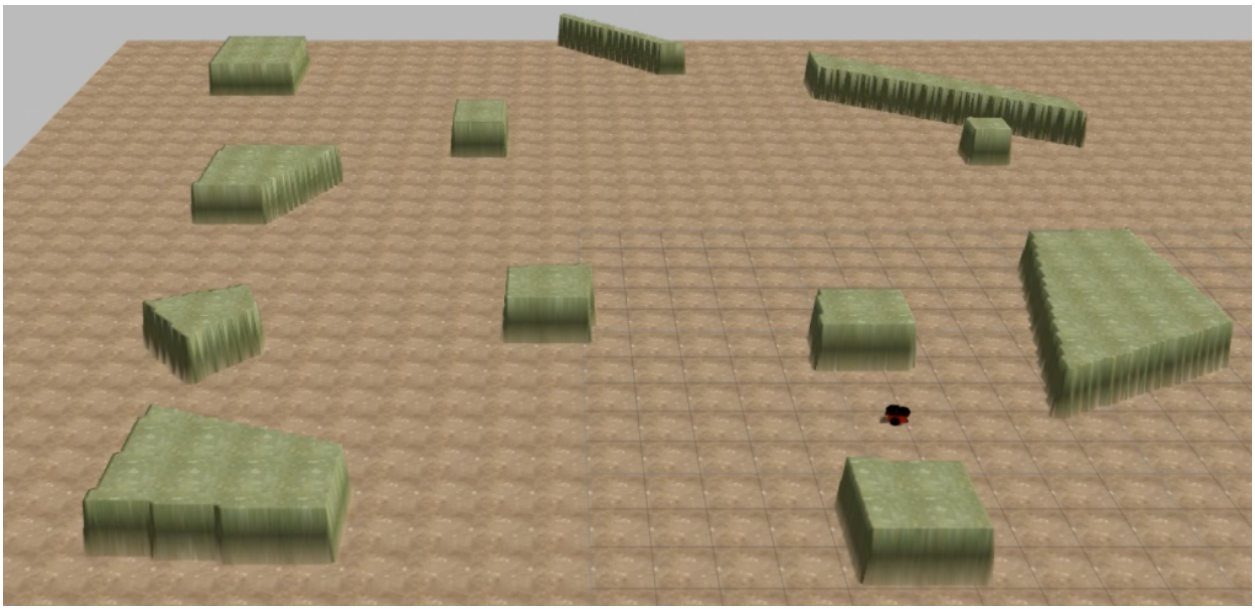


Figure 15: Testbed used.

A. Simulation Result, Method 1

The tesbed was constructed using the Gazebo software which was running in the Robot Operating System (ROS); the example used was a rover scenario, much like those given in.[22] Figure 15 shows a screenshot of the testbed used. The tasks chosen are *GOTO Waypoint* ω_1, *GOTO Waypoint* ω_2, and *GOTO Waypoint* ω_3.

The BT executed is depicted in Fig. 16, where ω_1, ω_2, ω_3 are three waypoints in the state space as shown in Fig. 17a. Note that more general tasks can be chosen; we chose navigation tasks for convenience.

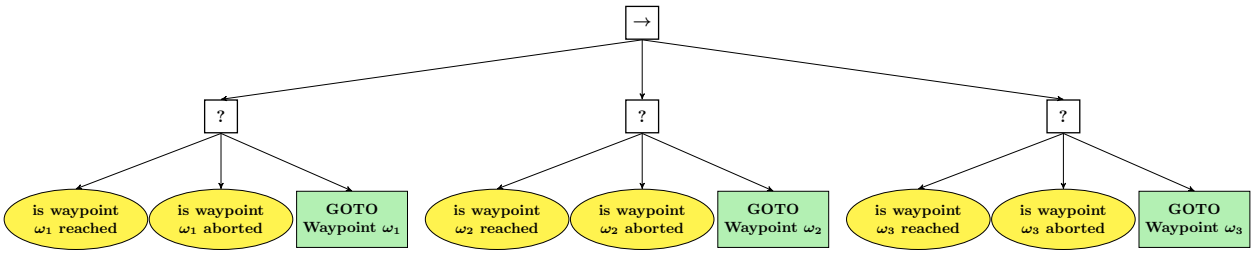


Figure 16: Proposed tree.

We ran Algorithm 1, setting some power constraints for each waypoint (Line 3). These constraints are translated into running regions, as depicted in Fig. 17a. Note that the success region of *GOTO Waypoint* ω_1 is not entirely inside the running region of *GOTO Waypoint* ω_2 (i.e., $x \in R_i \implies x \in S_{i-1}$ in Lemma 1 does not hold). Algorithm 1 sends the new success region down to the HL (Line 12), then Algorithm 1 runs the BT as long as there is a task to execute (Line 16). While the system is reaching the waypoint ω_2, we manually inject a fault into the system, then Algorithm 1 recomputes the running and success region of each subtree (the regions are smaller due to the fault, e.g. to move the system consumes more power) and sends them to the HL (Line 12). The new regions are depicted in Fig. 18a. After a few seconds we apply a second fault. Algorithm 1 recomputes the regions (Fig. 18b). Now the success region of *GOTO Waypoint* ω_2 is outside the running region of *GOTO Waypoint* ω_3 (i.e., $S_2 \cap R_3 = \emptyset$), hence the task *GOTO Waypoint* ω_2 is aborted (Line 15) and the system starts the execution of *GOTO Waypoint* ω_3.

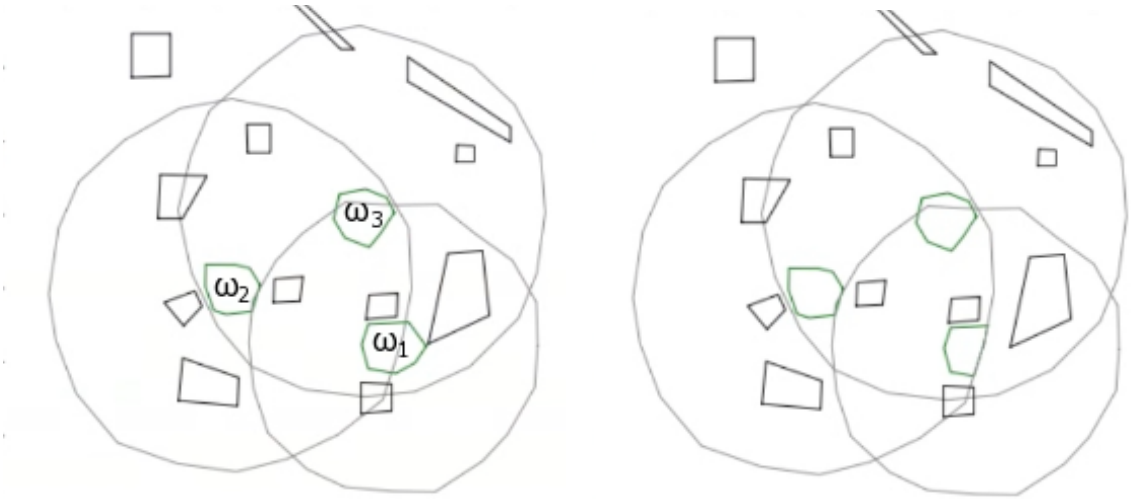
B. Interim Results, Method 2

We have begun initial testing of the optimization-based layering architectural analysis framework using a Dubins car model for a rover driving on a surface with high slippage / bad traction. (The original method that was tested used model predictive control (MPC) and PID control to encompass what we call the habitual and reflexive layers,[48] but preliminary testing showed that the MPC-PID tradeoff did not work well for our purposes. Thus, we moved on to using robust control theory for this approach, adding an additional feedback signal from the tracking layer to the planning layer.) Initial results for the method as described in section IV.B look promising, and will be discussed in a future paper, along with a longer discussion on the top-down functionality allocation that was done.

VI. Conclusion and Future Work

In this paper, we have given a description of total system stability, and techniques towards proving such, that we believe will help with the analysis and V&V of system architectures for robotics in the future, towards better enabling the adoption of risk-aware autonomy on future spacecraft. We have described two methods that are necessary (though not sufficient) towards giving guarantees of total system stability for a particular implementation of an RSE architecture using CSA-type modules. In future work, we will attempt to define sufficient (not just necessary) conditions for total system stability.

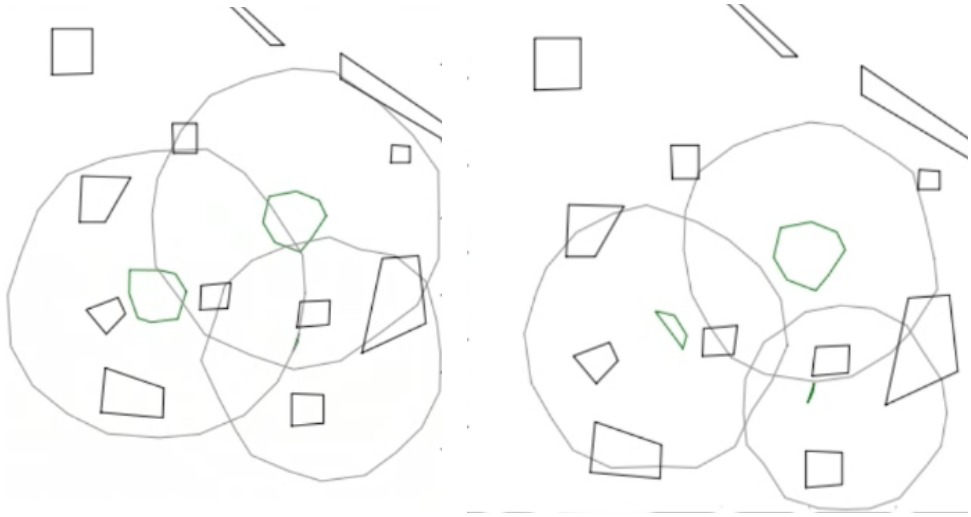
In future work, we also plan to further develop a third method of analysis that involves policy-checking, first for single-modules (all components inside a CSA module), and then for multiple modules (across multiple layers). To do this, we plan to use TuLiP, finite state machines, and LTL logic specifications to detail how to handle the problem of architecture-wide analysis at a high level, formally checking for issues like deadlock and whether timing constraints can be met for a given implementation. (We have already started this process, having hand-coded the policies for several CSA modules for an existing RSE implementation; these policies can easily be converted over for testing.) Once we have formal specifications for such problems in TuLiP, we can then 'reverse' the problem and expand upon existing methods for automatic verification of each module, to allow for the autocoding of the policies for the CSA components from these total system stability specifications.



(a) Original Success and Running regions.

(b) Constrained Success and Running regions.

Figure 17: Constraint passing from DL to HL. The success region for waypoint ω_1 is restricted to preserve TSS.



(a) Updated Success and Running regions after the first fault.

(b) Updated Success and Running regions after the second fault.

Figure 18: Updated regions during the execution. The success region for waypoint ω_2 is restricted to preserve TSS. The success region of GOTO Waypoint ω_2 is outside the running region of GOTO Waypoint ω_3.

Acknowledgments

The authors would like to thank the Model-based Embedded Robotic Systems Group at MIT for their input and feedback throughout the development process, especially Erez Karpas and Pedro Santana for all their help in answering our questions on the Enterprise system. The authors would also like to thank the Keck Institute of Space Studies for its initial study and final report on Engineering Resilient Space Systems, from which this effort had originated.

The research described in this paper was carried out at the Jet Propulsion Laboratory under a contract with the National Aeronautics and Space Administration, and at the California Institute of Technology under a grant from the Keck Institute for Space Studies.

References

[1]McGhan, C., Murray, R., Serra, R., Ingham, M., Ono, M., Estlin, T., and Williams, B., "A risk-aware architecture for resilient spacecraft operations," *Aerospace Conference, 2015 IEEE*, March 2015, pp. 1–15.

[2]Nayak, P. P., Bernard, D. E., Dorais, G., Kanefsky, E. B. G. J. B., Gamble, E. B., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Rajan, K., Rouquette, N., wen Tung, Y., Smith, B. D., and Taylor, W., "Validating The DS1 Remote Agent Experiment," 1999.

[3]Muscettola, N., Nayak, P. P., Pell, B., and Williams, B. C., "Remote Agent: To Boldly Go Where No AI System Has Gone Before," 1998.

[4]Brown, D., "NASA's Deep Impact Produced Deep Results," URL: `http://www.nasa.gov/mission_pages/deepimpact/media/deepimpact20130920f.html`, 2013.

[5]NASA Jet Propulsion Laboratory, "JPL — Missions — Deep Impact – EPOXI," URL: `http://www.jpl.nasa.gov/missions/deep-impact-epoxi`, 2014.

[6]Gray, D. L. and Brown, G. M., "Fault-Tolerant Guidance Algorithms for Cassini's Saturn Orbit Insertion Burn," *Proceedings of the American Control Conference (ACC 905)*, June 1998.

[7]Nesnas, I., Wright, A., Bajracharya, M., Simmons, R., Estlin, T., and Kim, W. S., "CLARAty: An architecture for reusable robotic software," *SPIE Aerosense Conference*, 2003.

[8]Nesnas, I. A., "CLARAty: A collaborative software for advancing robotic technologies," *Proceedings of NASA Science and Technology Conference*, Vol. 2, 2007.

[9]Chien, S., Sherwood, R., Tran, D., Cichy, B., Rabideau, G., Castano, R., Davis, A., Mandl, D., Trout, B., Shulman, S., and Boyer, D., "Using Autonomy Flight Software to Improve Science Return on Earth Observing One," *Journal of Aerospace Computing, Information, and Communication*, Vol. 2, No. 4, 2005, pp. 196–216.

[10]Ono, M. and Williams, B. C., "An Efficient Motion Planning Algorithm for Stochastic Dynamic Systems with Constraints on Probability of Failure," *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08)*, 2008.

[11]Ono, M., *Robust, Goal-directed Plan Execution with Bounded Risk*, Ph.D. thesis, Massachusetts Institute of Technology, 2012.

[12]Blackmore, L., Li, H., and Williams, B., "A probabilistic approach to optimal robust path planning with obstacles," *American Control Conference, 2006*, IEEE, 2006, pp. 7–pp.

[13]Ono, M. and Williams, B. C., "Iterative Risk Allocation: A New Approach to Robust Model Predictive Control with a Joint Chance Constraint," *Proceedings of 47th IEEE Conference on Decision and Control*, 2008.

[14]Ono, M., "Joint Chance-Constrained Model Predictive Control with Probabilistic Resolvability," *Proceedings of American Control Conference*, 2012.

[15]Ono, M., Williams, B., and Blackmore, L., "Probabilistic Planning for Continuous Dynamic Systems," *Journal of Artificial Intelligence Research*, Vol. 46, 2013, pp. 449–515.

[16]Jewison, C., BcCarthy, B., Sternberg, D., Fang, C., and Strawser, D., "Resource Aggregated Reconfigurable Control and Risk-Allocative Path Planning for On-orbit Assembly and Servicing of Satellites," *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, AAAI, 2014.

[17]Ono, M., Graybill, W., and Williams, B. C., "Risk-sensitive Plan Execution for Connected Sustainable Home," *Proceedings of the 4th ACM Workshop On Embedded Systems (BuildSys)*, 2012.

[18]Blackmore, L., *Robust Execution for Stochastic Hybrid Systems*, Ph.D. thesis, Massachusetts Institute of Technology, 2007.

[19]Wongpiromsarn, T., Topcu, U., and Murray, R. M., "Synthesis of Control Protocols for Autonomous Systems," Vol. 1, 2013, pp. 21–39.

[20]McGhan, C. L. and Murray, R., "Application of Correct-by-Construction Principles for a Resilient Risk-Aware Architecture," *AIAA SPACE 2015 Conference and Exposition*, 2015.

[21]Wongpiromsarn, T., Topcu, U., Ozay, N., Xu, H., and Murray, R. M., "TuLiP: A Software Toolbox for Receding Horizon Temporal Logic Planning," *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, HSCC '11, ACM, New York, NY, USA, 2011, pp. 313–314.

[22]McGhan, C. L. R., Vaquero, T., Subrahmanya, A. R. K. L., Arslan, O., Murray, R., Ingham, M. D., Ono, M., Estlin, T., Williams, B., and Elaasar, M., "The Resilient Spacecraft Executive: An Architecture
for Risk-Aware Operations in Uncertain Environments," *AIAA SPACE 2016 Conference and Exposition*, 2016. (Accepted).

[23]Murray, R. M., Day, J. C., Ingham, M. D., Reder, L. J., and Williams, B. C., "Engineering Resilient Space Systems: Final Report," *Keck Institute of Space Studies, Final Report*, , No. September, 2013, pp. 84.

[24]Burdick, J. W., du Toit, N., Howard, A., Looman, C., Ma, J., Murray, R. M., and Wongpiromsarn, T., "Sensing, Navigation and Reasoning Technologies for the DARPA Urban Challenge," *Technical report, DARPA Urban Challenge Final Report*, 2007.

[25]Wongpiromsarn, T. and Murray, R. M., "Distributed mission and contingency management for the DARPA Urban Challenge," *International Workshop on Intelligent Vehicle Control Systems, 2008*, IEEE, 2008, p. Submitted.

[26]Dvorak, D., Rasmussen, R. D., Reeves, G., and Sacks, A., "Software architecture themes in JPL's mission data system," *Proceedings of 2000 IEEE Aerospace Conference*, 2000.

[27]Rasmussen, R. D., "Goal based fault tolerance for space systems using the mission data system," *Proceedings of 2001 IEEE Aerospace Conference*, 2001.

[28]Barrett, A., Knight, R., Morris, R., and Rasmussen, R., "Mission planning and execution within the mission data system," *Proceedings of the International Workshop on Planning and Scheduling for Space*, 2004.

[29]Ingham, M., Rasmussen, R., Bennett, M., and Moncada, A., "Engineering complex embedded systems with state analysis and the mission data system," *Journal of Aerospace Computing, Information and Communication*, 2005.

[30]Colledanchise, M. and Ögren, P., "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture and Decision Trees," *IEEE Transactions on Robotics*, (Submitted).

[31]Millington, I. and Funge, J., *Artificial intelligence for games*, CRC Press, 2009.

[32]Rabin, S., *Game AI Pro*, chap. 6. The Behavior Tree Starter Kit, CRC Press, 2014.

[33]Florez-Puga, G., Gomez-Martin, M. A., Gomez-Martin, P. P., Diaz-Agudo, B., and Gonzalez-Calero, P. A., "Query-Enabled Behavior Trees," *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 1, No. 4, pp. 298–308.

[34]Colledanchise, M., Marzinotto, A., and Ögren, P., "Performance Analysis of Stochastic Behavior Trees," *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, June 2014.

[35]Colledanchise, M. and Ögren, P., "How Behavior Trees Generalize the Teleo-Reactive Paradigm and And-Or-Trees," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2016.

[36]Ögren, P., "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees," *AIAA Guidance, Navigation and Control Conference, Minneapolis, MN*, 2012.

[37]Klökner, A., "Interfacing Behavior Trees with the World Using Description Logic," *AIAA conference on Guidance, Navigation and Control, Boston*, 2013.

[38]Colledanchise, M. and Ögren, P., "How Behavior Trees Modularize Robustness and Safety in Hybrid Systems," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, June 2014.

[39]Nicolau, M., Perez-Liebana, D., O'Neill, M., and Brabazon, A., "Evolutionary Behavior Tree Approaches for Navigating Platform Games," *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. PP, No. 99, 2016, pp. 1–1.

[40]Hu, D., Gong, Y., Hannaford, B., and Seibel, E. J., "Semi-autonomous Simulated Brain Tumor Ablation with Raven II Surgical Robot using Behavior Tree," *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[41]Guerin, K. R., Lea, C., Paxton, C., and Hager, G. D., "A Framework for End-User Instruction of a Robot Assistant for Manufacturing," *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[42]Bagnell, J. A. D., Cavalcanti, F., Cui, L., Galluzzo, T., Hebert, M., Kazemi, M., Klingensmith, M., Libby, J., Liu, T. Y., Pollard, N., Pivtoraiko, M., Valois, J.-S., and Zhu, R., "An Integrated System for Autonomous Robotics Manipulation," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, October 2012, pp. 2955–2962.

[43]Klöckner, A., "Behavior Trees with Stateful Tasks," *Advances in Aerospace Guidance, Navigation and Control*, Springer, 2015, pp. 509–519.

[44]Chiang, M., Low, S. H., Calderbank, A. R., and Doyle, J. C., "Layering as optimization decomposition: A mathematical theory of network architectures," *Proceedings of the IEEE*, Vol. 95, No. 1, 2007, pp. 255–312.

[45]Seborg, D. E., Mellichamp, D. A., Edgar, T. F., and Doyle III, F. J., *Process dynamics and control*, John Wiley & Sons, 2010.

[46]Cai, D., Mallada, E., and Wierman, A., "Distributed optimization decomposition for joint economic dispatch and frequency regulation," *2015 54th IEEE Conference on Decision and Control (CDC)*, IEEE, 2015, pp. 15–22.

[47]El-Samad, H., Kurata, H., Doyle, J., Gross, C., and Khammash, M., "Surviving heat shock: control strategies for robustness and performance," *Proceedings of the National Academy of Sciences of the United States of America*, Vol. 102, No. 8, 2005, pp. 2736–2741.

[48]Matni, N. and Doyle, J. C., "A Theory of Dynamics, Control and Optimization in Layered Architectures," *2016 IEEE American Control Conference (ACC)*, 2016.

[49]Dahleh, M. and Pearson, J. B., "$\mathcal{L}_1$-optimal feedback controllers for MIMO discrete-time systems," *IEEE Transactions on Automatic Control*, Vol. 32, No. 4, 1987, pp. 314–322.

[50]Wang, Y.-S., Matni, N., and Doyle, J. C., "Localized Optimal Control Framework – Part I: Localizability Theory," *submitted to IEEE Transactions on Automatic Control*, 2016.

[51]Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y., "ROS: an open-source Robot Operating System," *ICRA Workshop on Open Source Software*, 2009.

[52]Crick, C., Jay, G., Osentoski, S., and Jenkins, O. C., "ROS and ROSbridge: Roboticists out of the Loop," *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction*, HRI '12, ACM, Boston, Massachusetts, USA, 2012, pp. 493–494, ISBN: 978-1-4503-1063-5.

[53]Adept MobileRobots, "MobileSim - MobileRobots Research and Academic Customer Support (release 0.7.3)," URL: `http://robots.mobilerobots.com/wiki/MobileSim`, 2014.