33rd Annual INCOSE international symposium
hybrid event
Honolulu, HI, USA
July 15 - 20, 2023

Autonomica: Ontological Modeling and Analysis of Autonomous Behavior

Maged Elaasar, Nicolas Rouquette, Klaus Havelund,
Martin Feather, Saptarshi Bandyopadhyay and Alberto Candela
{maged.e.elaasar, nicolas.f.rouquette, klaus.havelund,
martin.s.feather, saptarshi.bandyopadhyay, alberto.candela.garza}@jpl.nasa.gov
NASA Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA

Abstract. Model-based system autonomy is a complex integration of planning from high-level goals to low-level command sequences whose execution controls a system. The need for autonomy has accelerated in recent years to enable complex missions in automotive, space, and defense. During system development, understanding the relationship between system autonomy and the physical environment (including hardware) is critical to supporting trade studies, developing concepts of operations, characterizing risk, and performing testing. This paper describes the initial results of developing Autonomica, an ontology-based methodology and a framework for autonomous behavior modeling and analysis. This methodology formalizes an architectural pattern for specifying model-based autonomy as a vocabulary with description logic semantics and provides authoring and analysis capabilities (reasoning, querying, and simulation) for the architectures. The framework implements the methodology in an integrated workbench. A running example of a hypothetical spacecraft mission to a small space body illustrates the ideas.

1. Introduction

Autonomy is the ability of a system to achieve goals while operating independently of external control (Fong et al, 2018). The need for autonomy is increasing in many domains. In the automotive setting, an autonomous car navigates between two locations, without the control of a human driver, on roads along with vehicles, pedestrians, and other obstacles. In aerospace, a robotic spacecraft operates independently of ground-based control, with time delays and limited communication windows. In defense, an autonomous drone flies without a pilot in hostile airspace tracking its targets while avoiding being attacked. In all these cases, the system operates in challenging and dynamic environments characterized by large amounts of risk and uncertainty. The system resources (e.g., power, sensors, memory) may also be limited or degraded due to the harshness of the operating environment. Autonomy becomes increasingly critical to improve productivity, increase robustness, and eventually reduce cost

Architecting autonomous behavior is a complex endeavor that requires careful specification of the control system, the system under control (the hardware and operating environment), and their interactions. Within the control system, it involves specifying the autonomy functions of planning, scheduling, execution, and monitoring, in addition to traditional functions of estimation and control. Such specifications need a well-defined methodology that helps systems architects think about relevant details, make decisions, and document them in a coherent way that supports efficient analysis for consistency, completeness, and correctness.

State Analysis (SA) (Ingham et al, 2005) is an architectural pattern that was developed over a decade ago. It provides a useful vocabulary for specifying model-based autonomy (planning, scheduling, and execution), specifically putting emphasis on a clear definition of state and a clean separation of estimation and control. It defines the boundaries of the system under control and the control system and defines the interfaces between them. However, the pattern has not been formalized in a known language (such as e.g. UML), nor has it been incorporated into a modeling methodology. Another issue with SA is that its semantics is weakly formalized. We think that addressing these issues would allow SA to form the basis for an approach to autonomy modeling and analysis.

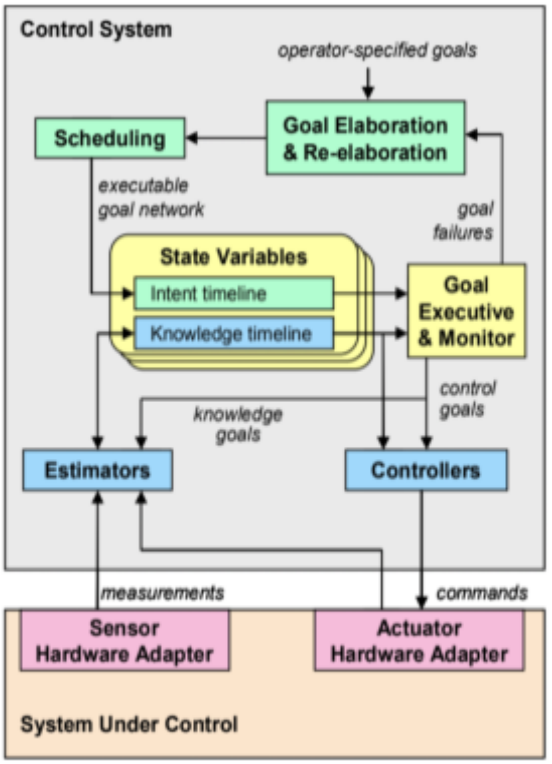


Figure 1. State Analysis Architectural Pattern

This paper introduces a new methodology and framework called Autonomica for autonomy modeling and analysis. The methodology formalizes SA as a vocabulary in the Ontological Modeling Language (OML) [openCAESAR], which has description logic (DL) semantics. This allows SA models to be easily checked for consistency using an off-the-shelf DL reasoner. It also allows encoding the well-formedness rules of SA using the SPARQL query language (SPARQL). Moreover, the SA vocabulary builds on a small set of foundational vocabularies for systems engineering called IMCE (Integrated Model Centric Engineering) [IMCE], which has been used in some space mission development. The methodology also identifies the steps of using the SA vocabulary and facilitates applying them using clear steps. Finally, the Autonomica framework supports the methodology by integrating some tools, including openCAESAR (openCAESAR) for ontological modeling and analysis, MEXEC (Verma et al, 2016) for task planning, scheduling, execution and monitoring, and Python/Matlab for simulation.

The rest of this paper is structured as follows. Section 2 overviews technologies used in this work. A running example used to illustrate the ideas is presented in Section 3. Section 4 describes the Autonomica methodology, including its usage to model all the relevant levels of autonomy. The Autonomica framework that implements the methodology is discussed in section 5. Section 6 overviews related works. Finally, Section 7 concludes and outlines future work.

2. Background

State Analysis. State Analysis (SA) (Ingham et al, 2005, Wagner et al, 2009) is an architectural pattern, illustrated in Figure 1 (from (Wagner et al, 2009)), for designing a Control System (CS) that controls a System Under Control (SUC). The SUC is typically hardware and/or a physical environment but can include software. The state analysis pattern promotes the following three main design principles. *(1) State as a first-order consideration:* The CS maintains a representation of (best estimate of) the state of the SUC with the concept of state variables, whose types are timelines. A timeline reflects knowledge of the state variable's past and current values as well as future projections. A state variable is analogous to a function of time calculating the past, current, or projected value based on available evidence. Note the important distinction between a state variable in the CS and the corresponding physical state of the SUC that it estimates. State variables can represent two kinds of timelines: intent and knowledge. Intent timelines represent the past,

present, and planned objectives from the schedule of operations perspective. Knowledge timelines represent the result of the estimation process based on all available evidence in the CS (i.e., goals, measurements, and commands). *(2) Separation of estimation and control:* this separation helps avoid design mistakes where control decisions are made on the basis of measurements instead of state estimates, resulting in complex and brittle control systems. Instead, estimators update knowledge timelines on the basis of all available evidence: goals, sensor measurements, and control commands, whereas controllers query knowledge timelines as needed to issue commands. Incidentally, the asymmetry in estimation vs. control reflects a common observation among feedback system designers where the crux of the complexity typically occurs in estimation instead of control. *(3) Goal-directed operation:* operator intent is expressed as goals rather than concrete commands (at the top level). A goal is a constraint on one or more timelines: what the values should be over a certain time period. This may optionally be augmented with additional quality and performance measures. Goals are elaborated into a schedule (a process also referred to as *planning* and *scheduling*), ultimately resulting in commands issued to the SUC and measurements obtained from the SUC for monitoring progress toward the eventual achievement of the goals. Goals can be control goals (to initiate control, usually referring to intent timelines) or estimation goals (to initiate estimation, usually referring to knowledge timelines). State analysis requires a single attribution of estimation and control: each state variable knowledge timeline is updated by a single estimator in the CS; and each actuator in the SUC receives commands from a single controller in the CS.

OML. The Ontological Modeling Language (OML) is used for describing knowledge as semantic ontologies with description logic (DL) semantics. OML allows defining two kinds of ontologies: vocabulary to define terms (concepts, properties, and relations) and inference rules of a given domain, and description that uses vocabularies to assert knowledge. OML ontologies can be checked for logical consistency using DL reasoners, which can also generate logical entailments from them. A dataset made of assertions and entailments can be loaded to a database and queried using the SPARQL (SPARQL) query language. In section 4, we use OML to formalize the SA vocabulary, then use that vocabulary to describe the autonomous behavior of the running example. We then analyze that behavior for consistency, query it to check well-formedness or gain insights, and generate Python/Matlab skeleton code from it. We use OML's canonical textual notation to describe the running example. We show the SA vocabulary using OML's graphical notation, which resembles that of UML class diagrams (or Entity-Relation diagrams) with some variations (extra annotations).

IMCE. The Integrated Model Centric Engineering (IMCE) vocabularies (openCAESAR) is a library of foundational vocabularies for system engineering that is defined in OML and has been used in several applications (Anonymous-1 2020). The library includes vocabularies like *base* (basic design patterns), *mission* (structural design patterns), and *analysis* (analysis design patterns). In section 4, we reuse these vocabularies in the definition of the SA vocabulary.

openCAESAR. The open-source project openCAESAR defines the OML specification and provides a reference implementation for it in Java. The project also provides an Eclipse-based authoring workbench for OML called Rosetta that allows the creation of OML vocabularies using its textual or graphical syntaxes. It also provides the ability to author OML descriptions. Rosetta also allows analyzing OML models for consistency and running SPARQL queries on them. In section 5, we describe how openCAESAR is used to develop the Autonomica framework.

MEXEC. The MEXEC (Multi-mission EXECutive) framework (Troesch et al, 2020) is used for scheduling task networks and subsequently executing the resulting schedules while monitoring their execution. A task network is defined in terms of tasks, which at the higher levels are defined as Boolean constraints, specifying their conditional impact on state variables (as in SA). At the lowest level, tasks are defined by commands to the system under control. By performing smart scheduling and execution, it is possible to react to failures, unexpected events, and execution

uncertainties. Current conditions (values of state variables) are used to adjust the schedule based on task specifications. The close coupling between scheduling of tasks and execution and monitoring of tasks, where the estimators and controllers update the state variables, results in a closed-loop control that adjusts schedules continuously.

3. Running Example: Small Body Mission

We ground our work in an example mission to a small body in space (Nesnas et al, 2021), and scope our effort on the approach phase (Figure 2), starting when the small body becomes visible in the spacecraft's cameras and ending when the spacecraft starts proximity operations.

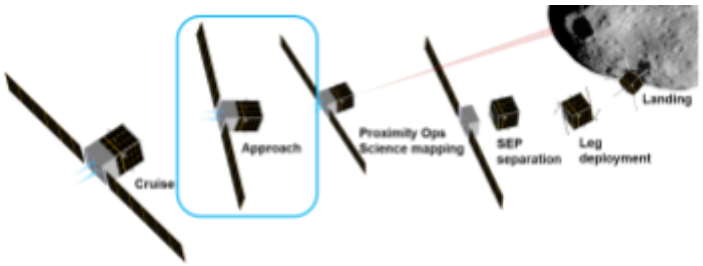


Figure 2. Small Body Mission Concept

We consider some tasks within this phase of the mission. The first task is *Communication with Earth (COMM*), which needs to occur on a regular schedule, predetermined by the limited availability of ground antennas to communicate with this mission. Each communication requires orienting the spacecraft antenna towards Earth, during which time its solar panels may be sub-optimally oriented towards the Sun. The net effects are a draw-down of battery power (needed by the radio), a transfer of data to Earth, and possibly an updated set of instructions from Earth. The second task is a *Trajectory Correction Maneuver (TCM)*, which also occurs on a regular schedule, predetermined to be able to keep the spacecraft on course towards the small body according to on-board optical measurements of the spacecraft's relative position and velocity with respect to the small body. A TCM may be skipped if the calculated spacecraft trajectory changes are small enough that, if applied, would inject undesirable noise instead of making significant progress. When a TCM is needed, it requires the spacecraft to be oriented in the direction that will provide the thrust vector in the appropriate direction; like a COMM, this may orient the solar panels away from the Sun. The net effects are a draw-down of battery charge (the thrusters must be electrically heated prior to operating), and a course correction. The third task is an Observation of the small body (SB), which is interspersed between the COMM and TCM tasks. The approach trajectory is designed so that the Sun is illuminating the small body from the spacecraft's point of view. The spacecraft's cameras face the small body, and solar panels face back toward the Sun. The net effects are battery charging, and accumulation of data - images of the small body - from which information about the small body (e.g., its spin rate) is determined.

In the approach phase of the mission, the spacecraft needs to manage resources carefully: First, the *Battery State of Charge* must be kept above a minimum threshold; once the battery is fully charged, excess power is automatically shunted to a radiator to dissipate it as heat. Second, the *Data Storage Space* reflects the accumulation of small body observations and the release of storage space during communication windows when observations are downlinked. If the storage is getting full, a policy determines which data to discard. Third, *Attitude Control* reflects the exclusive nature of the spacecraft's attitude direction requirements: towards the Sun for optimal power generation, towards Earth for communication, and towards the small body for observation. As described, a COMM, a TCM, and an SB each need attitude control, and the orientations they need are incompatible with one another. This means that those three kinds of tasks are mutually exclusive at any point in time.

Figure 3 shows a scenario, highly simplified for ease of presentation here. Time flows from left to right. The colored segments denote tasks scheduled for execution. Each TCM and COMM task starts at a fixed time (indicated by the push-pins), and each depletes the battery at some rate. Each SB starts after a COMM (which may finish ahead of time if there is little information to convey), and continues until the next TCM starts, during which time the battery is replenished, if possible up to its capacity (100%). The blue line indicates the projected state of charge of the battery, the green line the progress towards the small body, which is sped up by TCMs. Execution may not follow this projection exactly, and may deviate significantly in the event of a failure (e.g., failure of a solar panel). With this scenario, we could e.g. investigate how the spacecraft is approaching the small body under variations of the length and frequency of these tasks, and how they make changes to the relative distance from the spacecraft to the small body.

Figure 3. Example a task network of the running example and its impact on key state

4. Autonomica Methodology

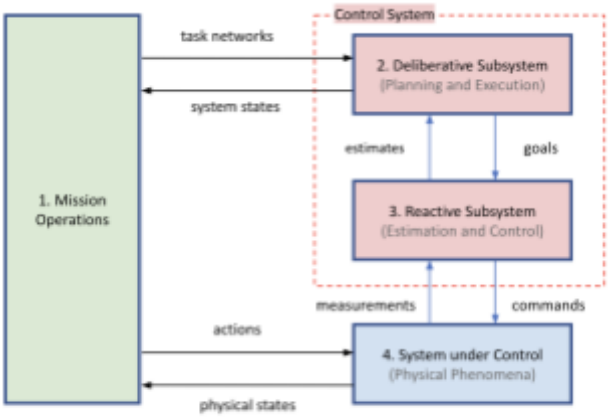


Figure 4. Autonomica Architecture

In this section, we describe the Autonomica methodology for modeling and analyzing autonomous behavior. The methodology adopts a refinement of SA as an architectural pattern and formalizes it with an OML vocabulary. The vocabulary allows describing the four layers of the pattern (Figure 4): the Mission Operations, (MO, section 4.1), the Deliberative Subsystem, (DS, section 4.2), the Reactive Subsystem, (RS, section 4.3) and the System Under Control (SUC, section 4.4).. The combination of RS and DS corresponds to Figure 1's CS, but following SA principles, make an important distinction between the role of State Variables in the two subsystems: in DS, State Variables tend to have low-complexity (discrete values or number ranges) with a rate of update suitable for planning and execution; in RS, State Variables may have high-complexity and domain-specific representation (such as quaternions in the Guidance and Navigation domain) with a rate of update compatible with the performance requirements of estimation and control functions. These fundamental differences in the roles of state variables motivate splitting the single Control System into two functionally distinct layers: DS & RS - Figure 4.

Inter-layer interactions are depicted as vertical exchanges in Figure 4 (interactions with the mission operations are shown as horizontal exchanges). In each subsection below, we present relevant parts of the SA vocabulary. We show how the vocabulary is used to describe corresponding aspects from the running example. We discuss some of the possible analyses that can be run on the models of each layer. These analyses are enabled thanks to the DL semantics of the SA vocabulary and the ability to query using SPARQL. We also discuss how a skeleton implementation of each layer is generated from the OML descriptions that could be completed and used for simulation.

4.1. Modeling the Mission Operations

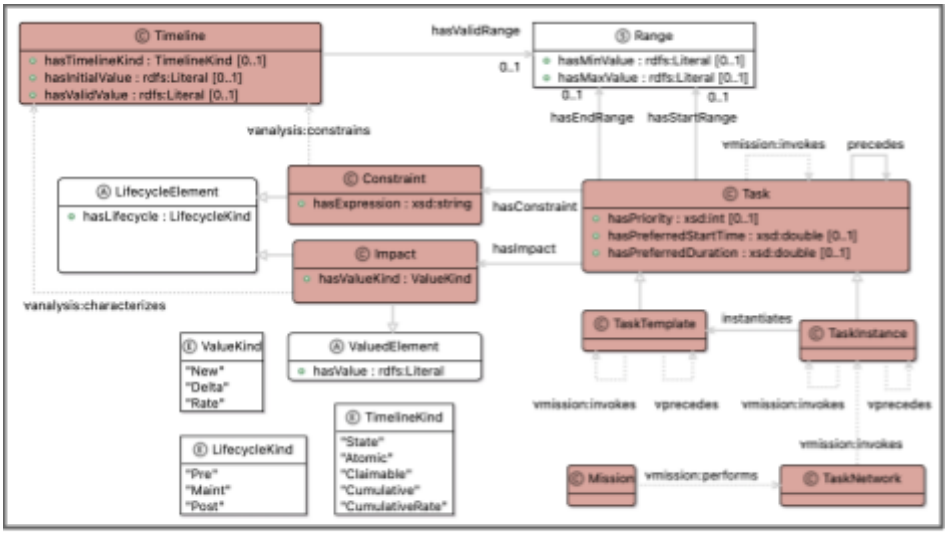


Figure 5. Subset of State Analysis vocabulary for Mission Operations

Vocabulary. Figure 5 shows the subset of the State Analysis vocabulary (in OML's graphical notation) supporting modeling Mission Operations. We discuss it below while italicizing the vocabulary terms we use. We also specify how it builds on the IMCE vocabularies (e.g., mission).

In this step, we formalize the scenarios a *Mission* performs as *Task Networks*, which invoke a set of *Tasks* performed by the autonomous system. A *TaskInstance* is a concrete *Task* that specifies a step in the scenario and may optionally instantiate (inherits a reusable set of specifications from) a *TaskTemplate*, which is like an abstract task. A *Task* is specified with a priority, a start range, an end range, a preferred start time, and a preferred duration. A *Task* is also specified in terms of a set of *Constraints* and/or *Impacts* it has on *Timelines*. A *Timeline* can be of kind *State* (assigned a value), *Atomic* (fully claimed in use), *Claimable* (partially claimed in use), *Cumulative* (assigned a value or claimed), or *Cumulative Rate* (assigned a value, claimed, or changed at a rate). A *Constraint* is a (pre, maintenance, or post) condition that a *Timeline* meets relative to the *Task*. An *Impact* represents a value to set, a *delta* to add, or a *rate* to add periodically to a *Timeline* relative to (*pre*: at the beginning, *maint*: during, or *post*: end of) a *Task*. An *Impact* gets used by the DS to estimate the future values of *Timelines*. A *Task* can be specified to invoke sub *Tasks* (hierarchical Tasks) and can be constrained to precede (have temporal dependency on) other *Tasks*.

Description. We use this subset of the SA vocabulary to describe the approach scenario of the case study mission (Figure 3) as shown in Table 11. We specify the scenario as a task network that consists of two task instances: TCM1, which instantiates the TCM (Trajectory Correction Maneuver) task template, and SBO1, which instantiates the SBO (Small Body Observation) task template and precedes TCM1. Both task instances specify maintenance impacts on a SC_SB_rel_distance (spacecraft to small body relative distance) timeline with values in the range 0% to 100%. (*Notice that "ci X : Y" means concept instance X is of type Y*).

Table 1. The mission operations in the running example

```
description <http://MO1#> as MO1 { uses <http://mds.jpl.nasa.gov/sa/state-analysis#> as SA
 ci SC_SB_rel_distance : sa:Timeline [ sa:hasTimelineKind "State" sa:hasValidRange [sa:hasMinValue 0; sa:hasMaxValue 100] ]
 ci TCM : sa:TaskTemplate
 ci TCM1 : sa:TaskInstance [ sa:instantiates TCM ]
 ri TCM1.impact : sa:HasImpact [from TCM1; to RS2:SC_SB_rel_distance;  sa:hasLifecycle "Maint"; sa:hasValueKind "Rate"; sa:hasValue -3]
 ci SBO : sa:TaskTemplate
 ci SBO1 : sa:TaskInstance [ sa:instantiates SB; sa:precedes TCM1 ]
 ri SBO1.impact : sa:HasImpact [ from SB1; to RS2:SC_SB_rel_distance; sa:hasLifecycle "Maint"; sa:hasValueKind "Rate" sa:hasValue 1 ]
 ci Approach : sa:TaskNetwork [ mission:invokes SB1; mission:invokes TCM1 ]
 ci SB_Mission : sa:Mission [ mission:performs Approach ]
}
```

4.2. Modeling the Deliberative Subsystem

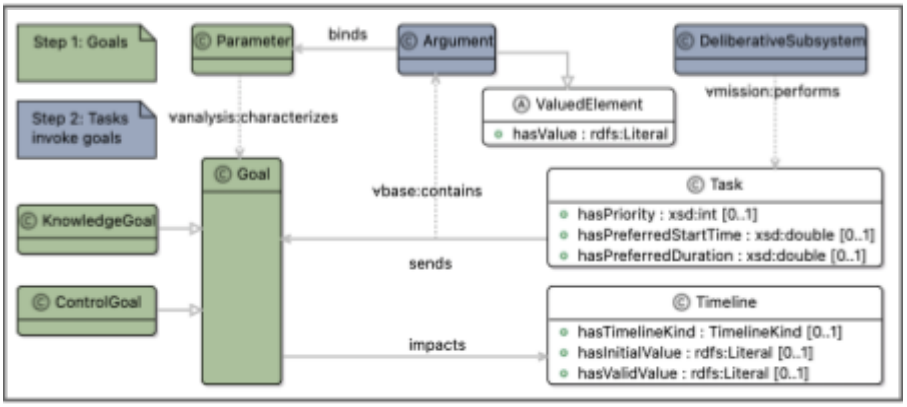


Figure 6. Subset of the State Analysis vocabulary for Deliberative Subsystem

Recall (from Figure 4) that a DS is the subsystem of the CS that receives from Mission Operations a task network representing tasks to perform. It plans, schedules and executes those tasks by sending goals to be achieved by the RS. It then monitors their achievement, detects any failures, and handles them by replanning. The following subset of the methodology helps model the DS.

Vocabulary. Figure 6 shows the subset of the State Analysis vocabulary (in OML graphical notation) supporting the two steps of the Autonomica methodology pertaining to modeling the DS. We discuss them below while italicizing the vocabulary terms we use. We also specify how it builds on the IMCE vocabularies (e.g., base and mission).

Step 1 involves defining *Goal*s, functional capabilities of the RS, invoked by the DS via *Tasks*. A *Goal* specifies which *Timeline*s it impacts and can be characterized with a set of *Parameters*. A *Goal* is either a *KnowledgeGoal*, which is about improving the knowledge of some Timeline, or a *ControlGoal*, which is about influencing the value of some Timeline.

Step 2 involves defining which *Goals* of the RS are invoked by *Task*s of the DS when they are executed. If a *Goal* has parameters, its invocation involves specifying values as *Arguments* bound to each *Parameter*. A *Goal* is invoked when its preconditions are met, after which the DS monitors its achievement (execution), through the *Task*'s maintenance and post conditions, and handles detected failures by replanning (we omit describing this for brevity).

Description. We use the above described subset of the vocabulary to specify in Table 2 the DS in the running example (Figure 3) and the RS *Goals* sent by each DS *Task*. We define four goals: a pair of knowledge and control goals on the distance to the small body, and a knowledge and control goal on the trajectory. (*Notice that keyword ref refers to previously defined instances*).

Table 2. The deliberative subsystem in the running example

```
description <http://DS1#> as DS1 { extends <http://MO1#> as MO1
  ci DS : sa:DeliberativeSubsystem [ mission:performs MO1:TCM1, MO1:SBO1 ]
  ref ci MO1:SB  [ sa:sends KnowDistance, ReduceDistance, KnowTrajectory, ImproveTrajectory ]
  ref ci MO1:TCM [ sa:sends KnowTrajectory; sa:sends ImproveTrajectory ]
  ci KnowDistance : sa:KnowledgeGoal [ sa:impacts MO1:SC_SB_rel_distance ]
  ci ReduceDistance : sa:ControlGoal [ sa:impacts MO1:SC_SB_rel_distance ]
  ci KnowTrajectory : sa:KnowledgeGoal [ sa:impacts MO1:SC_SB_rel_distance ]
  ci ImproveTrajectory : sa:ControlGoal [ sa:impacts MO1:SC_SB_rel_distance ]
}
```

Analysis. The subset of the SA vocabulary for describing the DS also allows answering some analytical questions. For example, one audit checks that the Timelines that are impacted by Tasks are also specified to be impacted by Goals invoked by those Tasks. This check can be done by running the SPARQL query shown in Table 3.

Table 3. A SPARQL query to find timelines impacted by tasks but no goals defined on them

```
PREFIX base: <http://imce.jpl.nasa.gov/foundation/base#> PREFIX sa: <http://mds.jpl.nasa.gov/state-analysis#>
SELECT DISTINCT ?task ?timeline WHERE {
 ?task a sa:Task ; sa:hasImpact [ analysis:characterizes ?timeline ]
  FILTER NOT EXISTS { ?goal sa:impacts ?timeline }
}
```

Code Generation. The SA model of the DS layer allows generation of an implementation skeleton in some executable language. In the running example, a Python implementation that uses the MEXEC planner (running as a ROS node) is generated. MEXEC uses an XML format for task networks that can directly be generated from the OML description (Table 2). Moreover, the description (in Table 3) can be used to generate Goals and Tasks classes. The Task classes create instances of Goal classes and publish them (to the RS) on ROS topics. They also subscribe to Timeline updates (from RS) used for planning, scheduling and monitoring.

4.3. Modeling the Reactive Subsystem

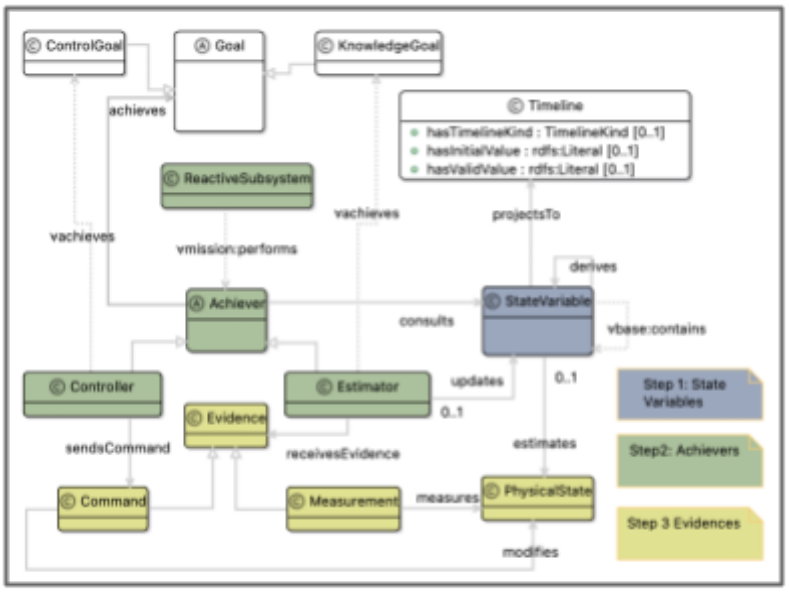


Figure 7. Subset of State Analysis vocabulary for Reactive Subsystem

Recall (from Figure 4) that an RS is a subsystem of the CS that interfaces with the DS subsystem to achieve the sent goals in a (relatively slow) closed loop. It achieves those goals by interfacing with the system under control (SUC) to perform estimation and control functions in (relatively fast) closed loops. This subset of the methodology is for modeling an RS.

Vocabulary. Figure 6 shows a subset of the State Analysis vocabulary (in OML graphical notation) supporting the 3 steps of the Autonomica methodology pertaining to RS modeling.

Step 1 involves defining *StateVariable*s, which on one hand are used to derive Timelines in the DS and on the other hand, represent estimated values of *PhysicalStates* in the SUC. Note that not all *PhysicalState*s in the SUC need to be estimated by the RS; only those that need to be consulted for control functions. Each *PhysicalState* is estimated by at most one *StateVariable*.

Step 2 involves defining the *Achiever*s of the RS and which *Goals* they achieve. Achievers receive *Goal*s from the DS and achieve them via closed-loop behavior that involves *consulting StateVariable*s. One kind of *Achiever* is an *Estimator* which achieves *KnowledgeGoal*s, and the other kind is *Controller*, which achieves *ControlGoal*s.

Step 3 involves defining the interface between the RS and the SUC layers. This involves specifying which *PhysicalStates* are estimated by the *StateVariables*, and how the estimation and control functions use them. Specifically, *Estimators* update *StateVariables* based on *receivingEvidence* on *PhysicalStates* such as *Measurement*s from SUC *Sensor*s and *Command*s from SUC *Actuator*s. Similarly, *Controllers* try to influence PhysicalStates by *sendingCommand*s to SUC *Actuator*s.

Description. We further elaborate on the example of Sec. 4.2 using the RS vocabulary defined above. Table 4 shows the results of the methodology's 1st step. For example, we define *StateVariables Camera.fieldOfView*, *Thruster.torque*, *Spacecraft.traj* and *Spacecraft-SB.rel-traj*. We then specify that the last variable projects to the SC-SB.rel_distance *Timeline*, and the first three variables together derive the last variable. Notice the use of relation instances (ri) for saying this.

Table 4. The state variables in the running example

```
description <http://RS1#> as RS1 { extends <http://MO1#> as MO1
  ci Camera.fieldOfView : sa:StateVariable
  ci Thruster.torque : sa:StateVariable
  ci Spacecraft.traj : sa:StateVariable
  ci Spacecraf-SB.rel_traj : sa:StateVariable
  ri Spacecraft-SB.rel_traj.projection : sa:ProjectsTo [ from Spacecraft-SB.rel_traj to MO1:SC-SB.rel_distance ]
  ri Spacecraft-SB.rel_traj.derivation : sa:Derives [ from Camera.fieldOfView Thruster.torque, Spacecraft.traj to Spacecraft-SB.rel_traj ]
}
```

In the 2nd step, Table 5 defines the *Achievers* (*Estimators* and *Controllers*) of the RS, which *KnowledgeGoals* / *ControlGoals* they achieve, and which StateVariables they consult / update.

Table 5. The intent state variables and goals in the running example

```
description <http://RS2#> as RS2 { extends <http://RS1#> as RS1, <http://DS#> as DS
  ci RS : sa:ReactiveSubsystem [ mission:performs DistanceE, DistanceC, TrajectoryE, TrajectoryC ]
  ci DistanceE : sa:Estimator [ RS; sa:achieves DS:KnowDistance;  sa:updates RS1:Spacecraft-SB.rel_traj; sa:consults RS1:Spacecraft.traj ]
  ci DistanceC : sa:Controller [sa:achieves DS:ReduceDistance; sa:consults RS1:Spacecraft-SB.rel_traj ]
  ci TrajectoryE : sa:Estimator [ sa:achieves DS:KnowTrajectory;  sa:updates RS1:Spacecraft.traj ]
  ci TrajectoryC : sa:Controller [ sa:achieves ImproveTrajectory; sa:consults RS1:Spacecraft.traj ]
}
```

Table 6 shows the results of the methodology's third step which defines the *PhysicalStates* (*Camera.fieldOfView*, *Thruster.torque_thrust*, and *Spacecraft.traj*) estimated by the StateVariables defined above. It also describes how the *Achievers* estimate them and influence them through *Measurements* and *Commands* (as *Evidences*).

Table 6. The achievers in the running example

```
description <http://RS3#> as RS3 {  extends <http://RS2#> as RS2, <http://RS1#> as RS1, <http://SUC1#> as SUC1
```

```
  ref ci Camera.fieldOfView [ sa:estimates SUC1:Camera.fieldOfView ]
  ref ci Thruster.torque_thrust [ sa:estimates SUC1:Thruster.torque_thrust ]
  ref ci Spacecraft.traj  [ sa:estimates SUC1:Spacecraft.traj ]
  ref ci DistanceE  [ sa:receivesEvidence SUC1:Camera.Image ]
  ref ci DistanceC [ sa:sendsCommand SUC1:Thruster.cmd ]
  ref ci TrajectoryE  [ sa:receivesEvidence SUC1:Trajectory.data, SUC1:Thruster.cmd ]
  ref ci TrajectoryC  [  sa:sendsCommand SUC1:Thruster.cmd ]
}
```

Analysis. The SA vocabulary for describing the RS supports answering analytical questions. For example, an Autonomica methodology audit checks that every physical state that is (directly or indirectly) controlled must also be estimated. Table 7 shows the SPARQL query for this audit.

Table 7. A SPARQL query to find all states that are controlled but not estimated

```
01  PREFIX sa: <http://mds.jpl.nasa.gov/state-analysis#>
02  SELECT ?state ?controller WHERE {
03          ?controller a sa:Controller ;  sa:sendsCommandTo/sa:affects ?state .
04          FILTER NOT EXISTS { ?state sa:isEstimatedBy/sa:isUpdatedBy ?estimator . }
05  }
```

Code Generation. The RS models allow generation of an implementation skeleton in a similar way to the DS (section 4.2) models. Table 8 shows a skeleton for a Matlab class of the Trajectory estimator whose function takes arguments according to all the evidence involved.

Table 8. Generated Matlab class representing the definition of a goal achiever

```
01  classdef TrajectoryE
02    methods
03      fun obj = receivesEvidence(obj, Spacecraft, Trajectory, Thruster)
04      % TODO: write estimation algorithm here
05      % inputs Thruster.cmd (or null if no data); Trajectory.data (or null if no data)
06      % output: may update Spacecraft.traj
07  end end end
```

4.4. Modeling the System Under Control

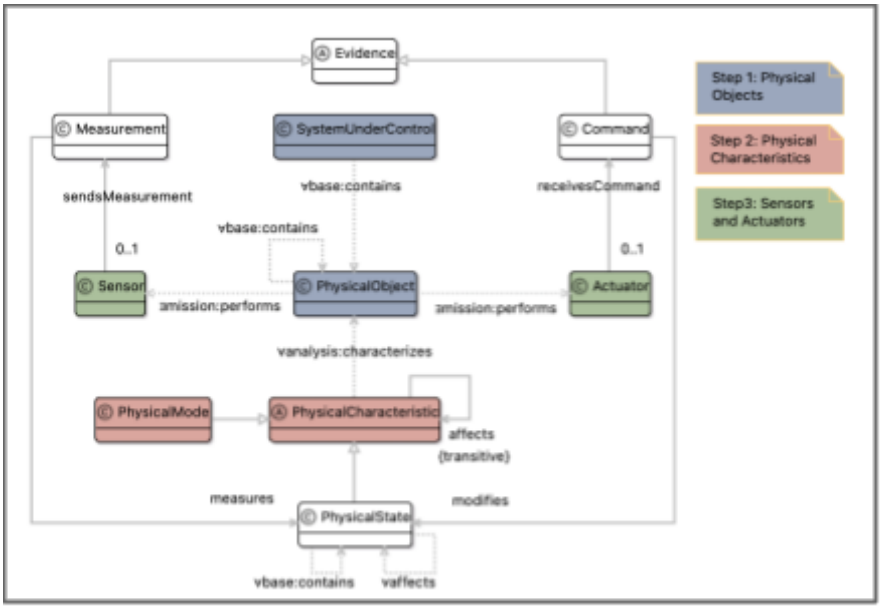


Figure 8. Subset of State Analysis vocabulary for System Under Control

Vocabulary. Figure 5 shows a subset of the State Analysis vocabulary (in OML's graphical notation) supporting the three steps of the Autonomica methodology that pertain to SUC modeling. We discuss those steps while referencing this figure. We italicize the vocabulary terms we use.

Step 1 involves defining the *PhysicalObject*s *contained* in the SUC, which are being controlled by the CS, and which typically include the hardware components of the system along with their *contained Sensor*s and *Actuator*s. We also identify other objects *contained* in the SUC whose state directly or indirectly affects or is affected by that of the former set of objects. These objects may either be other hardware components or objects in the physical (operational) environment.

Step 2 involves identifying the *PhysicalCharacteristics* that *characterize* the identified *PhysicalObject*s. It also involves specifying which *PhysicalCharacteristic*s directly affect others (i.e., may change when they change). Two kinds of *PhysicalCharacteristic*s can be defined: a *PhysicalMode* is a property of an object that can be changed instantly with an *Action* (e.g., a switch being on or off), whereas a *PhysicalState*, is a property that cannot be changed instantly with an Action (e.g., a battery charge level). Note that all the *PhysicalMode*s define an interface (of *Actions* that can modify those modes) presented by the SUC to mission operations. The SUC can also send to mission operations a snapshot of the current characteristics of the object. Such an interface allows mission operations to manipulate the SUC during simulation (e.g. a test engine can call those actions to challenge a CS, a feature that we plan to leverage in future works.)

Step 3 involves defining the interface between the SUC and the RS (Figure 4). Specifically, this involves defining the *Measurement*s that the *Sensor*s in the SUC can occasionally send to the RS. It also involves the *Command*s that *Actuators* in the SUC occasionally receive from the RS.

Description. With the vocabulary subset defined above, we can model a simplified version of the navigation domain aspects of the Approach scenario of Figures 3. Table 9 shows the results of the 1st sub step, separating the description of the SUC between the Environment (limited to the small body, SB, in this example) and the Spacecraft with its sensors and actuators.

Table 9. The physical objects in the running example

```
01  description <http://SUC1#> as SUC1 {
02    ci SUC : sa:SystemUnderControl
03    ci Environment : sa:PhysicalObject  [ base:isContainedIn SUC ]
04    ci SB : sa:PhysicalObject  [ base:isContainedIn Environment ]
05    ci Spacecraft : sa:PhysicalObject [ base:isContainedIn SUC ]
06    ci Camera : sa:Sensor [ mission:isPerformedBy Spacecraft ]
07    ci Trajectory : sa:Sensor [ mission:isPerformedBy Spacecraft ]
08    ci Thruster : sa:Actuator [ mission:isPerformedBy Spacecraft ]
09  }
```

Scoping the physical objects help designers identify the relevant physical characteristics and their affects relationships as described in Table 10 according to the 2nd sub step[1].

Table 10. The physical states in the running example

```
01  description <http://SUC2#> as SUC2 { extends <http://SUC1#> as SUC1
02    ci Camera.FieldOfView : sa:PhysicalState [ analysis:characterizes SUC1:Camera ]
03    ci Thruster.torque_thrust : sa:PhysicalState [ analysis:characterizes SUC1:Thruster; sa:affects Spacecraft.traj ]
04    ci Spacecraft.traj : sa:PhysicalState [ analysis:characterizes SUC1:Spacecraft; sa:affects Spacecraft-SB.rel_traj ]
05    ci Spacecraft-SB.rel_traj : sa:PhysicalState [ analysis:characterizes SB; sa:affects Camera.FieldOfView ]
06  }
```

Next, designers can shift to defining the interfaces between the SUC and the RS in terms of measurements and commands as described in Table 11 according to the methodology's 3rd step.

[1] Abbreviations: traj = trajectory (attitude, angular velocity, position and velocity); rel_traj = relative trajectory.

Table 11. The physical interfaces in the running example

```
description <http://SUC3#> as SUC3 { extends <http://SUC2#> as SUC2; extends <http://SUC1#> as SUC1
  ci Camera.Image : sa:Measurement [ sa:measures SUC2:Camera.FieldOfView; sa:isSentBySensor SUC1:Camera ]
  ci Trajectory.data : sa:Measurement [ sa:measures SUC2:Spacecraft.traj; sa:isSentBySensor SUC1:Trajectory ]
  ci Thruster.cmd : sa:Command [ sa:modifies SUC2:Spacecraft.traj; sa:isReceivedByActuator SUC1:Thruster ]
}
```

Analysis. The subset of the SA vocabulary for describing the SUC allows for answering some analytical questions. For example, it allows querying for all the *PhysicalState*s that could directly or indirectly be *affected* by *PhysicalMode*s. Table 12 shows a SPARQL query that encodes this.

Table 12. A SPARQL query to find all physical states potentially affected by physical modes

```
PREFIX sa: <http://mds.jpl.nasa.gov/state-analysis#>
SELECT ?mode ?state WHERE {
  ?mode a sa:PhysicalMode ; sa:affects ?state .
  ?state a sa:PhysicalState .
}
```

Code Generation. While the OML description models above capture the structure of the SUC, the behaviors of how the physical states change over time can be modeled in a computational implementation language like Matlab. The methodology supports this paradigm via code generation of Matlab Object Oriented classes corresponding to PhysicalObjects and of Matlab class method signatures corresponding to the causality of the effects relationships among PhysicalStates and PhysicalModes of PhysicalObjects. For example, Table 13 shows a Matlab function signature generated for computing the Camera.FieldOfView physical state.

Table 13. Excerpts of Matlab classes for computing the camera's field of view of the small body

```
classdef Camera
  properties
    FieldOfView
  end
  methods
    function obj = compute_FieldOfView(obj, SB)
    % TODO: write the Camera.FieldOfView calculation as a function of the
inputs.
    % inputs: SB.rel_traj
end end end
```

```
classdef SB
  properties
    rel_traj
end end
```

5. Autonomica Framework

In section 4, we described the Autonomica methodology and its SA vocabulary that we designed with OML. In this section, we describe the Autonomica Framework, a set of tools that we package in a modeling workbench to help designers apply the Autonomica methodology for modeling their autonomy architecture (section 5.1), analyze it for consistency (section 5.2), run queries on it (section 5.3), and generate an implementation skeleton from it (section 5.4).

5.1 Modeling Support

Thanks to the SA vocabulary being formalized as an OML vocabulary, the modeling of an autonomous mission, including its SUC, RS, DS, and MO layers can be formalized as OML descriptions (ontologies). We used an Eclipse-based OML workbench, called Rosetta, from the openCAESAR project to author the OML ontologies. Rosetta provides an OML text editor that can be used for that. Rosetta also provides default diagram-based and form-based editors as alternative viewpoints for authoring OML models.

5.2 Consistency Analysis

Thanks to the SA vocabulary being formalized in OML, which has DL semantics, it is possible to use a DL reasoner to check the consistency of OML descriptions. What is checked is that the descriptions are consistent with the semantics of the vocabulary and do not contain logical contradiction. Such a contradiction may either be asserted directly in the description or inferred by the DL reasoner based on the semantics of the vocabulary. For example, a designer may assert that a physical state is measured by two sensors (in the SUC), each of which sends a measurement that is received by a different estimator (in the RS). This contradicts the SA semantics that the relationship from an estimator to an estimated state is *inverse functional*. A DL reasoner can detect such a contradiction and even provide the designer with an explanation in the form of a minimal set of assertions and rules that lead to it.

5.3 Query Support

The Rosetta workbench allows a designer to conveniently author and run a set of SPARQL queries on an OML description and report results back in various formats (e.g., JSON, XML). We used this ability to author SELECT queries to extract information from the description for audits (e.g., Table 4) or insights (Table 9).

5.4 Code Generation.

We added a feature to the Rosetta workbench allowing generating the skeleton code of an implementation architected with SA. We generate for each layer an implementation in a high-level programming language. For the SUC and RS, we generate Matlab code that contains the expected data flows along with stubs for functions that need to be implemented directly in Matlab. Since layers of an SA-based architecture need to communicate with each other asynchronously via message passing (e.g. sensors in the SUC publish measurements that are received by estimators in the RS), we plan to make our implementations target the ROS [ROS] platform, which provides pub/sub infrastructure for message passing. Another feature of interest in ROS is that it has a global clock that synchronizes executions across the layers.

6. Related Work

Model-based systems engineering (MBSE) (Ramos et al, 2011) promotes the formalized application of modeling for describing systems. An example of that is the use of the Systems Modeling Language (SysML) to specify the system's requirements, structure, behavior, and parametrics using a set of standard viewpoints (diagrams and tables). A subset of the language has been given execution semantics, which helps directly simulate behavior using the model. However, the computational expressiveness of SysML is limited compared to that of other languages like Matlab or Modelica. Furthermore, SysML does not prescribe a particular modeling methodology. The Architecture and Analysis Definition Language (AADL) [Feiler et al, 2013] is an industry standard language for modeling real-time embedded systems. It is distinguished by its emphasis on strong (although semi-formal) semantics, which has motivated its use in projects emphasizing formal methods. The formalism supports definition of software or hardware components, with ports linked together with communication channels. It was originally developed for embedded avionics systems, and does not provide direct support for specification of autonomy architectures.

ROSPlan is a framework providing tools for AI Planning in a ROS system (Cashmore et al, 2015). It has a variety of nodes which encapsulate planning, problem generation, and plan execution. Applications include short-term human-robot interaction (Sanelli et al, 2017) and opportunistic planning in autonomous underwater vehicle (AUV) missions (Cashmore et al, 2018). The NASA-funded autonomous science rover project Toolbox for Research and Exploration (TREX), is

investigating techniques designed to improve operational efficiency and science yield of future lunar rover missions (Anonymous-2 2020). The autonomy framework is implemented in ROS. The approach employs science hypotheses, and high-level goals provided by scientists to a rover. The rover performs domain-specific planning and execution to modify its mission plan based on the data collected and how it supports the goals (Anonymous-3 2020). The framework has been deployed on an analog rover in several sites of geologic interest in the United States. (Castano et al, 2022) investigates the problem of operations for autonomy, that is, identifying interfaces, tools, and workflows required to effectively operate future highly autonomous spacecraft. This work uses an ad hoc ROS environment integrated with MEXEC to simulate an autonomous spacecraft and its operations for a flyby mission to the Neptune-Triton system. The focus is on operations for autonomous systems rather than on developing architectures for them.

For leads to more work on architectures for autonomous spacecraft, see (Tipaldi & Glielmo, 2017), which surveys model-based techniques and describes operational concepts for mission planning and execution in European space projects; and (Cividanes et al, 2019), which is a more recent and extensive survey of spacecraft on-board planning and scheduling, listing many examples.

The autonomy architecture we use is compatible with the guidance espoused in the Framework for Robust Execution and Scheduling of Commands On-Board, FRESCO [Amini et al, 2021]. FRESCO specifies guiding principles, functions, interfaces, and interactions from which mission-specific autonomous control architectures can be derived.

7. Conclusions and Future Work

This paper reports on our initial efforts to define the Autonomica methodology and implement its framework. The methodology adopts SA as an architectural pattern and formalizes it as an OML vocabulary (ontology) with DL semantics. The result is a precise SA syntax for describing an autonomy architecture whose logical semantics can be checked for consistency and can facilitate analysis with SPARQL queries. We used such query capabilities to formalize audits for SA that a user can run to detect methodological issues with their autonomy description. We also developed a code generator from SA descriptions that can produce a (canonical) implementation skeleton that can be completed by a developer manually in a high-level language (we used Matlab).

We plan to continue developing the Autonomica methodology and framework. Our goal is to streamline the development process of autonomy based on SA, including allowing the development of the architecture in a continuous integration fashion, where the architecture is version controlled in a repo and automatically checked on change. We plan to produce architecture views to facilitate the peer review process. On the implementation front, we plan to improve the generated skeleton by targeting ROS (as described in section 5.4). We also like to investigate and potentially use more advanced features of the MEXEC planner to support more elaborate planning (e.g., contingencies). We are also investigating adding more behavioral specifications to the SA description model to help generate better skeleton implementation code and control logic. We plan to analyze an SA-based autonomy implementation both statically and dynamically. Static analysis of the implementation code and the OML model would extract topological information views that should be consistent even if modeling and implementation progress concurrently. Dynamic checking would involve testing the implementation driven by insights from querying the SA model. We expect this to allow a test engine to focus the testing effort and make it more efficient.

Acknowledgment

The research described here was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

References

Amini, R, Fesq, L, Mackey, R, Mirza, F, Rasmussen, R, Troesch, M & Kolcio, K 2021. 'FRESCO: A Framework for Spacecraft Systems Autonomy', *2021 IEEE Aerospace Conf.* (50100) (pp. 1-18).

Cashmore, M, Fox, M, Long, D, Magazzeni, D, Ridder, B, Carrera, A, Palomeras, N, Hurtos, N, & Carreras, M 2015. 'ROSPlan: Planning in the Robot Operating System'' *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*. 2015..

Cashmore, M, Fox, M, Long, D, Magazzeni, D, and Ridder, B 2018. 'Opportunistic Planning in Autonomous Underwater Missions', *IEEE Transactions on Automation Science and Engineering*, vol. 15, no. 2, pp. 519-530, April 2018, doi: 10.1109/TASE.2016.2636662.

Castano, R, Vaquero, T, Rossi, F, Verma, V, Van Wyk, E, Allard, D, Huffmann, B, Murphy, EM, Dhamani, N, Hewitt, RA, Davidoff, S, Amini, R, Barrett, A, Castillo-Rogez, J, Choukroun, M, Dadaian, A, Francis, R, Gorr, B, Hofstadter, M, Ingham, M, Sorice, C, & Tierney, I 2022. 'Operations for Autonomous Spacecraft', *2022 IEEE Aerospace Conf.*, pp. 1-20,

Cividanes, F, Ferreira, M & Kucinskis, F 2019, 'On-board Automated Mission Planning for Spacecraft Autonomy: A Survey', *IEEE Latin America Transactions*, 17(06), pp.884-896.

Executable UML - Semantics of a Foundational Subset for Executable UML Models, Version 1.5, OMG. <https://www.omg.org/spec/FUML/1.5/About-FUML>, April 2021.

Feiler, PH & Gluch, DP 2013. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley (2013).

Fong, TW, Frank, JD, Badger, JM, Nesnas, IA, and Feary, MS 2018. 'Autonomous systems taxonomy', *Autonomous Systems CLT Meeting* (No. ARC-E-DAA-TN56290).

Ingham, MD, Rasmussen, RD, Bennett, MB & Moncada, AC 2005. 'Engineering complex embedded systems with state analysis and the mission data system', *Journal of Aerospace Computing, Information, and Communication*, 2(12), pp.507-536.

Nesnas, IA, Hockman, BJ, Bandopadhyay, S, Morrell, BJ, Lubey, DP, Villa, J, Bayard, DS, Osmundson, A, Jarvis, B, Bersani, M & Bhaskaran, S 2021. 'Autonomous Exploration of Small Bodies Toward Greater Autonomy for Deep Space Missions', *Frontiers in Robotics and AI*, 8.

OOSEM - Object-Oriented SE Method. INCOSE. <https://www.incose.org/incose-member-resources/working-groups/transformational/object-oriented-se-method>

openCAESAR - openCAESAR Enables Rigorous Systems Engineering Practice. California Institute of Technology <https://www.opencaesar.io/>

Ramos, AL, Ferreira, JV & Barceló, J 2011. 'Model-based systems engineering: An emerging approach for modern systems', *IEEE Transactions on Systems, Man, and Cybernetics*, Part C (Applications and Reviews), 42(1), pp.101-111.

ROS - Robot Operating System. <https://www.ros.org/>

Sanelli, V, Cashmore, M, Magazzeni, D, & Iocchi, L 2017. 'Short-Term Human-Robot Interaction through Conditional Planning and Execution', *Proceedings of the International Cont. on Automated Planning and Scheduling (ICAPS)*, 27(1), 540-548.

SPARQL - SPARQL 1.1 Query Language. WC3. <https://www.w3.org/TR/sparql11-query/>

SysML - OMG System Modeling Language - <https://www.omg.org/spec/SysML>

Tipaldi, M & Glielmo, L 2017 'A survey on model-based mission planning and execution for autonomous spacecraft', *IEEE Systems Journal*, 12(4), pp.3893-3905, 2017.

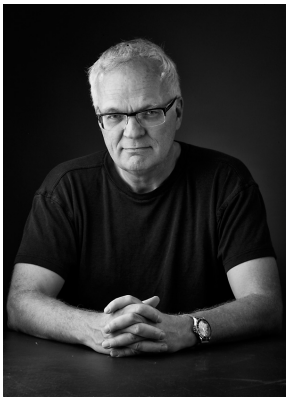
Verma, V, Gaines, D, Rabideau, G, Joshi, R & Schaffer, S 2016. 'MEXEC: autonomous science restart for the Europa mission'.

Wagner, D, Dvorak, D, Baroff, LE, Bennett, MB, Ingham, MD, Mittman, DS & Mishkin, AH 2009. 'A Control Architecture for Safe Human-Robotic Interactions During Lunar Surface Operations', *AIAA Infotech at Aerospace Conf.*, Seattle, Washington, 2009.

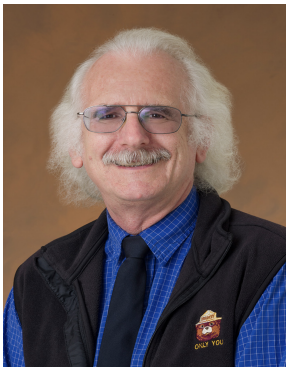
Biography

Maged Elaasar is a Senior Technology Researcher at the NASA Jet Propulsion Laboratory, California Institute of Technology, where he leads R&D projects in Model Based Systems Engineering and Autonomy. Specifically, Maged leads the Integrated Model Centric Engineering program, the openCAESAR project, and the Autonomica project. Prior to that, he was a senior software architect at IBM, where he led the Rational Software Architect family of modeling tools. He holds a Ph.D. in Electrical and Computer Engineering and M.Sc. in Computer Science from Carleton University, (2012, 2003), and a B.Sc. in Computer Science from American University in Cairo (1996). Maged is also the founder of Modelware Solutions, a technology consultancy and training company. He is also a lecturer in the department of Computer Science at the University of California Los Angeles.

Nicolas Rouquette is a Principal Computer Scientist at the Jet Propulsion Laboratory where his pioneering work on comprehensive code generation for JPL's Deep Space One mission paved the way for applying such techniques since then. He made key contributions to several revisions of modeling standards developed by the Object Management Group (OMG) for UML and SysML, including for UML 2.4.1 which became the first revision of UML for which formal verification of consistency was performed by mapping to OWL 2 and for SysML 1.4 which became the first revision of SysML for which comprehensive support for the 3rd revision of the Vocabulary of International Metrology (VIM) is available in SysML's ISO 80000 library. He holds a PhD in Computer Science from the University of Southern California where he enjoyed guidance on applied graph algorithms from Pavel Pevzner and then a PostDoc at USC's Math department. He also holds an engineering diploma from the French ESIEE engineering school in Paris.



Klaus Havelund is a Principal and Senior Research Scientist in the Flight Software and Avionics Systems Section at JPL. He has a Masters and PhD in Computer Science from the University of Copenhagen. He has worked at JPL since 2006 and before that at NASA Ames Research Center since 1997. His research is focused on formal methods. He has in particular over the last decades contributed to the runtime verification field. He has authored 154 papers.



Martin Feather is a Principal Software Assurance Engineer in JPL's Office of Safety and Mission Success. His focus is on research to assure space missions, in particular their software. A recipient of a NASA Exceptional Achievement Medal, he has been an author on over 180 publications spanning a range of topics, with the common theme of viewing the assurance problem from the perspective of what to be concerned about, and how to show those concerns are either absent or adequately addressed. He received his BA and MA in Mathematics and Computer Science from Cambridge University, UK, and PhD in Artificial Intelligence from the University of Edinburgh, UK.

Saptarshi Bandyopadhyay is a Robotics Technologist at the NASA Jet Propulsion Laboratory, California Institute of Technology, where he develops novel algorithms for future multi-agent and swarm missions. In 2020, he was named a NASA NIAC fellow for his work on the Lunar Crater Radio Telescope on the far-side of the Moon. He received his Ph.D. in Aerospace Engineering in 2016 from the University of Illinois at Urbana-Champaign, USA, where he specialized in probabilistic swarm guidance and distributed estimation. He earned his Bachelors and Masters degree in Aerospace Engineering in 2010 from the Indian Institute of Technology Bombay, India. His engineering expertise stems from a long-standing interest in the science underlying space missions, since winning the gold medal for India at the 9th International Astronomy Olympiad held in Ukraine in 2004. Saptarshi's current research interests include robotics, multi-agent systems and swarms, dynamics and controls, estimation theory, probability theory, and systems engineering. He has published more than 40 papers in journals and refereed conferences.

Alberto Candela is a Data Scientist in the Artificial Intelligence Group at the NASA Jet Propulsion Laboratory, California Institute of Technology. His research is focused on creating new autonomous science techniques for remote sensing, mobile robots, and spacecraft. To this end, Alberto works at the intersection of machine learning, planning, and decision making. Alberto's algorithms have been deployed on rovers and drones in remote locations around the world, including the Martian-like Atacama Desert. He received his Ph.D. and M.S in Robotics from Carnegie Mellon University, and his B.S. in Mechatronics Engineering from Instituto Tecnológico Autónomo de México (ITAM). After arriving at JPL, he has worked on pointing planning for instruments on board Earth-observing missions, spacecraft autonomy for small body mission concepts, and advanced machine learning for imaging spectroscopy, particularly for the EMIT mission.