Chapter 2
Handling Autonomy Requirements for ESA Systems

Abstract Contemporary software-intensive systems, such as modern spacecraft and unmanned exploration platforms (e.g., ExoMars) generally exhibit a number of autonomic features resulting in complex behavior and complex interactions with the operational environment, often leading to a need for self-adaptation. To properly develop such systems, it is very important to properly handle the autonomy requirements. This chapter discusses the notion of autonomy in the context of ESA Missions, and outlines aspects of requirements engineering along with specification models and formal methods for aerospace. The chapter goes in-depth about special *generic autonomy requirements* for space missions along with controller architectures for robotic systems controlling such missions. In detail are discussed formal methods and approaches that cope with both generic autonomy requirements and controller architectures, and as such can lay the foundations of a new Autonomy Requirements Engineering Model dedicated to autonomic features of space missions.

2.1 Introduction

Appropriate formal methods help ESA developers express and understand autonomy requirements to some extent. Formal methods assist in the construction of the Autonomy Requirements Engineering Model (AREM) suitable for the development of autonomous components for ESA systems. AREM takes into account all the aspects of an autonomic system as shown in Chap. 1 and emphasizes the so-called *self-* requirements* (Sects. 1.2.3 and 1.3) by taking into consideration the traditional *functional* and *non-functional* requirements of spacecraft systems (e.g., safety requirements). However, we can ask the question: *Why formal methods?* Traditionally, formal methods have had the necessary potential for modeling and validating the control behavior of software-intensive systems[1] and they may help in expressing autonomy requirements and modeling autonomic and self-adaptive behavior. It is our understanding that the application of formal methods will help ESA developers *unambiguously express autonomy requirements*, which are currently expressed in

[1] Modern spacecraft and autonomous robotics systems are considered to be software-intensive.

E. Vassev and M. Hinchey, *Autonomy Requirements Engineering for Space Missions*,
NASA Monographs in Systems and Software Engineering,
DOI 10.1007/978-3-319-09816-6_2

natural language. We expect that appropriate formal methods will improve the software development cycle of autonomic features for ESA's software-intensive systems in terms of:

- rigorous and unambiguous specification of autonomy requirements;
- autonomy requirements traceability, verification and validation;
- derivation of test cases and automatic test case generation based on requirements specification.

Moreover, a successful AREM should consider so-called *controller architectures for robotic systems* (Sect. 2.3) to eventually derive successor architectures for controllers for autonomous spacecraft.

Autonomy Requirements Engineering (ARE) should be considered as a software engineering process of (1) determining what *autonomic features* are to be developed for a particular software-intensive system or subsystems and (2) the software artifacts generated by that process. Note that the outcome of ARE (requirements specifications, models, etc.) is a precursor of design of autonomic features. The ARE process should involve all of the following:

- autonomy requirements elicitation;
- autonomy requirements analysis;
- autonomy requirements representation;
- autonomy requirements communication;
- development of acceptance criteria and procedures for autonomy requirements.

Note that the targeted AREM approach is a framework incorporating formal methods dedicated to autonomic features of software-intensive systems. The AREM framework allows for specification and modeling of autonomy requirements and it provides for validation and traceability of specified autonomy requirements. Thus, AREM is a requirements engineering approach helping to create reliable software that maximizes the probability of satisfying user expectations. This is possible because the framework toolset is going to provide verification mechanisms for *automatic reasoning* about specified autonomy requirements. A basic validation approach could be *consistency checking* where autonomy requirements are verified by performing exhaustive traversal to check for both syntax and consistency errors and to check whether requirements conform to predefined autonomy correctness properties, defined by ESA engineers. For example, correctness properties can be set to target the requirements feasibility.

Moreover, to handle *logical errors* (specification flaws) and to be able to assert safety (e.g., freedom from deadlock) and liveness (nice-to-have) properties, AREM can eventually provide for both model-checking and test-case generation mechanisms. Finally, AREM can be supplied with code generation mechanisms to facilitate the implementation of autonomic features.

2.1.1 Autonomy and Automation

Recall that automated processes replace routine manual processes with software/hardware ones that follow a step-by-step sequence that may still include human participation. *Autonomous processes*, on the other hand, emulate human processes rather than simply replacing them.

Complete autonomy may not be desirable or possible for some systems. In such cases, *adjustable* and *mixed autonomy* may need to be used [36]. In adjustable autonomy, the level of autonomy of the system (e.g., spacecraft) can vary depending on the circumstances or the needed interaction and control. The autonomy can be adjusted to be either *complete*, *partial*, or no *autonomy*. In these cases the adjustment may be done automatically by the system depending on the situation (e.g., an autonomous spacecraft may ask for help from mission control) or may be requested by the human control. Challenges in adjustable autonomy include knowing when it needs to be adjusted, as well as how much and how to make the transition between levels of autonomy. In mixed autonomy, autonomous agents and people work together to accomplish a goal or perform a task. Often agents perform the low level details of the task (e.g., analogous to the craft's preparation for landing) while the human performs the higher-level functions (e.g., analogous to the actual landing).

2.1.2 Levels of Autonomy for ESA Missions

ESA considers four *autonomy levels* for the execution of nominal mission operations [14]:

- execution mainly under real-time ground control;
- execution of pre-planned mission operations onboard;
- execution of adaptive mission operations onboard;
- execution of goal-oriented mission operations onboard.

These autonomy levels are summarized in Table 2.1. As shown in that table, ESA approaches the *autonomicity problem* very carefully in a stepwise manner. In this approach the highest-possible autonomy is the goal-oriented autonomy (level E4) where goals are determined by human operators and autonomous spacecraft decide what to do to autonomously achieve the desired goals. Still, this autonomy level hasn't been achieved yet. The current level of spacecraft autonomy is level E2 and ExoMars is expected to operate at level E3.

Table 2.1 ESA's Mission-execution autonomy levels [14]

Autonomy level	Description	Functions
E1	Mission execution under ground control Limited onboard capability for safety issues	Real-time control from ground for nominal operations Execution of time-tagged commands for safety issues
E2	Execution of pre-planned, ground-defined, mission operations onboard	Capability to store time-based commands in an onboard scheduler
E3	Execution of adaptive mission operations onboard	Event-based autonomous operations Execution of onboard operations control procedures
E4	Execution of goal-oriented mission operations onboard	Goal-oriented mission replanning

2.2 Requirements Engineering, Specification Models and Formal Methods for Aerospace

Requirements engineering is currently identified as one of the weak points of the software development process used in aerospace projects. Many space project reviews identify weakness in the software requirements in the early development [17]. Requirements solicitation, analysis, and management are key elements of a successful and safe development process for any system. Many costly and critical system failures can ultimately be traced back to missing, incorrect, misunderstood, or incompatible requirements. This leads to incomplete development and difficulties in system integration and software re-engineering and a very high price.

In Sect. 2.3.1, we discuss the characteristics of the Space Missions Requirements Analysis activity helping to identify classes of requirements for space missions.

2.2.1 Requirements Specification and Modeling

The importance of having consolidated software requirements makes the use of special *specification and modeling techniques* that help software engineers to achieve complete and consistent requirements desirable. Models at software level are called specifications (e.g., technical specification) and they may assists with the verification of the requirements and eventually with further design, implementation, and testing. For example, specification models may help software engineers with automatic code and test-case generation [16]. Of course, this approach requires a good definition of

the system/software process, a deep knowledge of the code generator, and a strong control of the software architecture.

For more on requirements engineering (e.g., requirements metrics and characteristics of requirements), please refer to Chap. 1.

2.2.2 Requirements Engineering for Autonomous Systems

An autonomous system is able to monitor its behavior and eventually modify the same according to changes in the operational environment, thus being considered as self-adaptation (Sect. 2.2.3 and for more details Chap. 1). As such, autonomous systems must continuously monitor changes in its context and react accordingly. But what aspects of the environment should such a system monitor? Clearly, the system cannot monitor everything. And exactly what should the system do if it detects less than optimal conditions in the environment? Presumably, the system still needs to maintain a set of high-level goals that should be satisfied regardless of the environmental conditions, e.g., mission goals of unmanned spacecraft used for space exploration. But non-critical goals could be not that strict [47], thus allowing the system a degree of flexibility during operation. These questions (and others) form the core considerations for building autonomous systems.

Traditionally, requirements engineering is concerned with *what a system should do and within which constraints it must do it* (Sect. 2.2). Requirements engineering for autonomous systems (and self-adaptive systems), therefore, must address what adaptations are possible and under what constrains, and how those adaptations are realized. In particular, questions to be addressed include: (1) *What aspects of the environment are relevant for adaptation*? and (2) *Which requirements are allowed to vary or evolve at run-time, and which must always be maintained?*. Requirements engineering for autonomous systems must deal with uncertainty, because the execution environment often is dynamic and the information about future execution environments is incomplete, and therefore the requirements for the behavior of the system may need to change (at run-time) in response to the changing environment.

Requirements engineering for autonomous systems appears to be a wide open research area with only a limited number of approaches yet considered. In this chapter, we present a few formal methods that eventually can be successful in capturing autonomy requirements.

2.2.3 Generic Autonomy Requirements

The first step towards development of a new software-intensive system is to determine the system's requirements, which includes both elicitation and specification (or modeling) of the same. In general, requirements fall into two categories: *functional* and *non-functional*. Whereas the former define the system's functionality

the latter emphasize system's qualities (e.g. performance) and constraints under which a system is required to operate. Like any computer system, *autonomic systems*[2] (ASs) also need to fulfill specific requirements from these two categories. However, unlike the other systems, the development of an AS is driven by the so called *self-management objectives* (also could be considered as *self-adaptive objectives*) and *attributes* (Chap. 1), which introduce special requirements termed self-* requirements [57]. Despite their differences in terms of application domain and functionality, all ASs are capable of self-management and are driven by one or more self-management objectives. Note that this requirement automatically involves (1) self-diagnosis (to analyze a problem situation and to determine a diagnosis) and (2) self-adaptation (to repair the discovered faults). The ability to perform adequate self-diagnosis depends largely on the quality and quantity of its knowledge of its current state, i.e., on the system awareness.

The following is a list of generic autonomy requirements [57] stemming from the self-* requirements:

2.2.3.1 Self-* Requirements (Autonomicity)

Autonomicity is one of the essential characteristics of ASs. Autonomicity aims at freeing human operators from complex tasks, which typically require a lot of decision making without human intervention. Autonomicity, however, is not only intelligent behavior but also an organizational manner. Adaptability is not possible without a certain degree of autonomy. A rule engine obeying a predefined set of conditional statements (e.g., if-then-else) put in an endless loop is the simplest form of autonomicity implementation. In many cases though, such a simple rule-based mechanism may not be sufficient and the rule engine should force feedback learning and learning by observation to refine the decisions concerning the priority of services and their granted objectives and quality of service, respectively.

2.2.3.2 Knowledge

Knowledge is a large complex aggregation composed of constituent parts representing knowledge of different kind. Every kind of knowledge may be used to derive knowledge models of specific domains of interest. For example, the following kinds of knowledge may be considered [13]:

- *domain knowledge*—refers to the application domain facts, theories, and heuristics;
- *control knowledge*—describes problem-solving strategies, functional models, etc.;
- *explanatory knowledge*—defines rules and explanations of the system's reasoning process, as well as the way they are generated;

[2] The term "autonomic systems" is often used in the scientific literature as a synonym of self-adaptive and autonomous systems.

- *system knowledge*—describes data contents and structure, pointers to the implementation of useful algorithms needed to process both data and knowledge, etc. System knowledge also may define user models and strategies for communication with users.

Moreover, being considered as essential system and environment information, knowledge may be classified as (1) internal knowledge—knowledge about the system itself and (2) external knowledge—knowledge about the system environment. Another knowledge classification could consider a priori knowledge (knowledge initially given to a system) and experience knowledge (knowledge gained from analysis of tasks performed during the lifetime of a system). Therefore, it depends on the problem domain what kinds of knowledge may be considered and what knowledge models may be derived from those kinds. For example, we may consider knowledge specific to:

- internal component structure and behavior;
- system-level structure and behavior;
- environment structure and behavior;
- different situations where an AS component or the system itself might end up in;
- components' and system's capabilities of communication and integration with other systems.

2.2.3.3 Awareness

An aware system is able to notice a change and understand the implications of that change. Conceptually, awareness is a product of *knowledge representation*, *knowledge processing*, and *monitoring*. In general, we address two types of awareness in ASs:

- *self-awareness*—a system (or a system's component) has detailed knowledge about its own entities, current states, capacity and capabilities, physical connections and ownership relations with other systems in its environment;
- *context-awareness*—a system (or a system's component) knows how to negotiate, communicate and interact with its environment and how to anticipate environmental states, situations and changes.

2.2.3.4 Monitoring

Monitoring is the process of obtaining knowledge through a collection of sensors instrumented within the system itself. Note that monitoring is not responsible for diagnostic reasoning or adaptation tasks. One of the main challenges of monitoring is to determine which information is most crucial for analysis of the system's behavior, and when. The notion of monitoring is closely related to the notion of awareness

because it is a matter of awareness, which information indicates a situation in which a certain adaptation is necessary.

2.2.3.5 Adaptability

Adaptability may result in changes to some functionality, algorithms or system parameters as well as the system's structure or any other aspect of the system. Note that self-adaptation requires a model of the system's environment. Adaptability is conceptualized as a concept to achieve change. It is in sharp contrast to creating new builds. A key research gap in this area is how to measure "adaptability".

2.2.3.6 Dynamicity

Dynamicity shows the system's ability to change at runtime. Dynamicity may also include a system's ability to exchange certain (defective or obsolete) components without changing the observable behavior. Conceptually, dynamicity deals with concerns like preserving states during functionality change, starting, stopping and restarting system functions, etc.

2.2.3.7 Robustness

ASs should benefit from robustness since this may facilitate the design of system parts that deal with the self-* requirements. Beside a special focus on error avoidance, several requirements aiming at correcting errors should also be enforced. Robustness can often be achieved by decoupling and asynchronous communication, e.g., between interacting AS components. Error avoidance, error prevention, and fault tolerance are proven techniques in software engineering, which help us in preventing error propagation when designing ASs.

2.2.3.8 Resilience

Adaptability might be considered as a quality attribute that is a prerequisite for resilience and system agility [28]. Closely related to safety, resilience enables aerospace systems to bounce back from unanticipated disruptions as well as to equip aging systems with the ability to respond to changing operational requirements.

2.2.3.9 Mobility

Mobility enfolds all parts of the system: from mobility of code on the lowest granularity level via mobility of services or components up to mobility of devices or even

mobility of the overall system. Mobility enables dynamic discovery and usage of new resources, recovery of crucial functionalities, etc. For example, ASs may rely on mobility of code to transfer some functionality relevant for security updates or other self-management issues.

2.3 Generic Autonomy Requirements for Space Missions

In this section, along with the Space Mission Requirements Analysis we elaborate on different classes of space missions and derive generic autonomy requirements per class of missions. Thus, we put in the context of space missions the generic autonomy requirements presented in Sect. 2.2.3.

2.3.1 Space Mission Requirements Analysis

Space Mission Analysis is an activity that takes aspects such as payload operational requirements and spacecraft system constraints as inputs, and generates as an output a mission specification. A key aspect of this process is the selection of the orbital parameters (or trajectory parameters) of the final mission orbit as well as the intermediate orbits during the early orbit acquisition phase. Note that the mission specification leads to design requirements on the spacecraft systems and subsystems. The Space Mission Analysis and Design (SMAD) Process consists of the following steps [20, 61]:

- Define Objectives:
  - Define broad objectives and constraints.
  - Estimate quantitative mission needs and requirements.
- Characterize the Mission:
  - Define alternative mission concepts.
  - Define alternative mission architectures.
  - Identify system drivers for each architecture.
  - Characterize mission concepts and architectures.
- Evaluate the Mission:
  - Identify critical requirements.
  - Evaluate mission utility.
  - Define baseline mission concept.
- Define Requirements:
  - Define system requirements.
  - Allocate requirements to system elements.

Typical Functional requirements are related to:

- *performance*: factors impacting this requirement include the primary mission objective, payload size, orbit, pointing;
- *coverage*: impacting factors include orbit, number of satellites, scheduling;
- *responsiveness*: impacting factors include communications architecture, processing delays, operations;
- *secondary mission* (if applicable).

Typical Operational requirements are:

- *duration*: factors impacting this requirement include nature of the mission (experimental or operational), level of redundancy, orbit (e.g., altitude);
- *availability*: impacting factors include level of redundancy;
- *survivability*: impacting factors include orbit, hardening, electronics;
- *data distribution*: impacting factors include communications architecture;
- *data content, form and format*: impacting factors include user needs, level and place of processing, payload.
- *ground station visibility*;
- *eclipse duration*: consider the eclipse period for spacecraft in an Earth orbit;
- *launch windows*: the time of launch of a spacecraft is often constrained by dynamic aspects related to reaching the mission orbit, or by system requirements;
- *communication windows*.

Typical Constraints are:

- *cost*: factors impacting this constraint include number of spacecraft, size and complexity, orbit;
- *schedule*: impacting factors include technical readiness, program size;
- *political*: impacting factors include Sponsoring organization (customer), whether international program;
- *interfaces*: impacting factors include level of user and operator infrastructure;
- *development constraints*: impacting factors include Sponsoring organization.

In general, space missions can be classified into two main groups: *Earth-orbiting missions* and *interplanetary missions* [20]. In the following sections, classes and subclasses of space missions are presented together with *generic autonomy requirements* for each one of these classes and subclasses. Note that the autonomy requirements presented here are currently not all realistic. Autonomy in space missions is about delegating control from the ground base to spacecraft, and due to security and cost reasons, neither ESA nor NASA is currently considering autonomy to take place in the processes of establishing missions' orbits or trajectories. Although not being realistic at the moment, we elaborate on such autonomy requirements, for the reason of completeness.

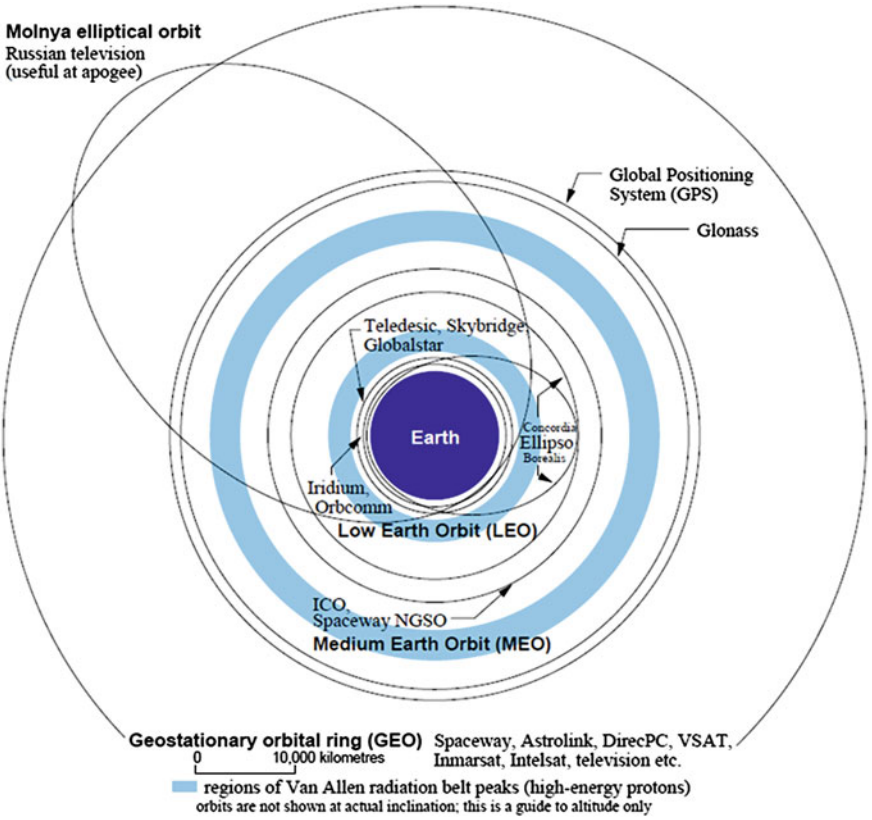

Fig. 2.1 Common earth orbits, figure courtesy of Wood [62]

2.3.2 Earth-Orbiting Missions

The Earth-Orbiting Missions is a class of missions that represents artificial satellites placed into Earth orbit and used for a large number of purposes. Different orbits give satellites different vantage points for viewing Earth, i.e., different Earth orbits give satellites varying perspectives, each valuable for different reasons. Some satellites hover over a single spot, providing a constant view of one face of the Earth, while others circle the planet. Figure 2.1 depicts common Earth satellite orbits [62].

Common Challenge: Orbital Perturbations

There are a variety of effects that will cause orbital perturbations during the lifetime of a satellite [5]:

- *third body perturbations*: dominated by the gravitational forces of the Sun and the Moon;
- *perturbations due to a non-spherical Earth*;

- *atmospheric drag*: principal non-gravitational force acting on a satellite and it only affects satellites in low-Earth orbit. Drag acts in the direction of opposite of the satellite's velocity resulting in a removal of energy from the orbit. This loss of energy results in the size of the orbit decreasing, which then leads to a further increase in drag;
- *solar radiation*: solar radiation pressure is an effect that is strongest on satellites with large area to mass ratios. It results in periodic variations in all orbital elements.

2.3.2.1 Polar Low Earth Orbit/Remote-Sensing Satellite Missions

These missions involve satellites that fly low orbit and use different earth-observation instruments that gather information about the Earth (land, water, ice and atmosphere) using a variety of measurement principles. The choice of orbit for a Low Earth Orbit (LEO) remote sensing spacecraft is governed by mission objectives and payload operational requirements. A LEO orbit is below an altitude of approximately 2,000 km (1,200 mi). Spacecraft in LEO encounter *atmospheric drag* in the form of gases in the thermosphere (approximately 80–500 km up) or exosphere (approximately 500 km and up), depending on orbit height. LEO is an orbit around Earth between the atmosphere and below the inner Van Allen radiation belt. The altitude is usually not less than 300 km because that would be impractical due to the larger atmospheric drag.

Equatorial low Earth orbits (ELEO) are a subset of LEO. These orbits, with low inclination to the Equator, allow rapid revisit times and have the lowest $\triangle V$ (a measure of the amount of "effort" that is needed to change from one trajectory to another by making an orbital maneuver) requirement of any orbit. Orbits with a high inclination angle are usually called polar orbits. Higher orbits include Medium Earth Orbit (MEO), sometimes called intermediate circular orbit (ICO), and further above, Geostationary Orbit (GEO).

Mission Challenges and Generic Autonomy Requirements

The common challenge in Polar LEO and Remote-Sensing Satellite Missions is to determine the right orbit altitude. The orbit altitude is principally established by a trade-off between instrument resolution and the fuel required to maintain the orbit in the presence of aerodynamic drag. Orbits higher than low orbit can lead to early failure of electronic components due to intense radiation and charge accumulation.

Considering these issues, we determine the following *autonomy requirements*:

- *self-* requirements* (autonomicity):
  - *self-orbit* (autonomously acquire the target orbit; adapt to orbit perturbations);
  - *self-protection* (autonomously detect the presence of radiation and move to escape);
  - *self-scheduling* (based on operational goals and knowledge of the system and its environment, autonomously determine what task to perform next);

  - *self-reparation* (implies operations re-planning based on performance degradation or failures);

- *knowledge*: mission objectives, payload operational requirements, instruments onboard together with their characteristics (e.g., instruments resolution), the Van Allen radiation belt, ground stations, communication links, data transmission format, orbit planes, eclipse period, spacecraft altitude, communication mechanisms onboard, Earth gravity;
- *awareness*: orbit awareness, radiation awareness, altitude awareness, position awareness, instrument awareness, neighboring satellites, sensitive to thermal stimuli, Earth gravitational force, data-transfer awareness, ground station visibility awareness, Earth rotation awareness, speed awareness, communication awareness, altitude awareness, air resistance awareness;
- *monitoring*: electronic components, surrounding environment (e.g., radiation level), atmospheric drags, ground station, altitude and orbit;
- *adaptability*: adaptable mission parameters, adapt to loss of energy, adapt to high radiation, adapt to weak satellite-ground station communication link, adapt to low energy;
- *dynamicity*: dynamic communication links;
- *robustness*: robust to temperature changes, robust to orbital perturbations, robust to communication losses;
- *resilience*: loss of energy is recoverable, resilient to radiation;
- *mobility*: information goes in and out, changing position within the orbit plane.

2.3.2.2 Satellite Constellation Missions

These missions are presented by multi-satellite systems where a group of satellites called a "constellation" work together. Such a constellation can be considered to be a number of satellites with coordinated ground coverage, operating together under shared control, synchronized so that they overlap well in coverage and complement rather than interfere with other satellites' important coverage [20]. For a constellation to operate, it may be necessary to use more than a single ground station, especially when the space segment consists of a large number of satellites (Fig. 2.2). Inter-satellite links (ISL) are bidirectional communication links between satellites in LEO or MEO orbits.

Mission Challenges and Generic Autonomy Requirements

There are a few important challenges in Satellite Constellation Missions. Such missions rely on high distributiveness: a distributed system in space and a distributed system on ground are combined in a distributed space mission. One of the major issues is that the topology of the distributed space mission changes over time, which places stringent requirements on communication. The topology change is on the one side caused from the orbit dynamics, on the other side may be manually controlled to switch to a desired formation or constellation. Moreover, due to the movement of

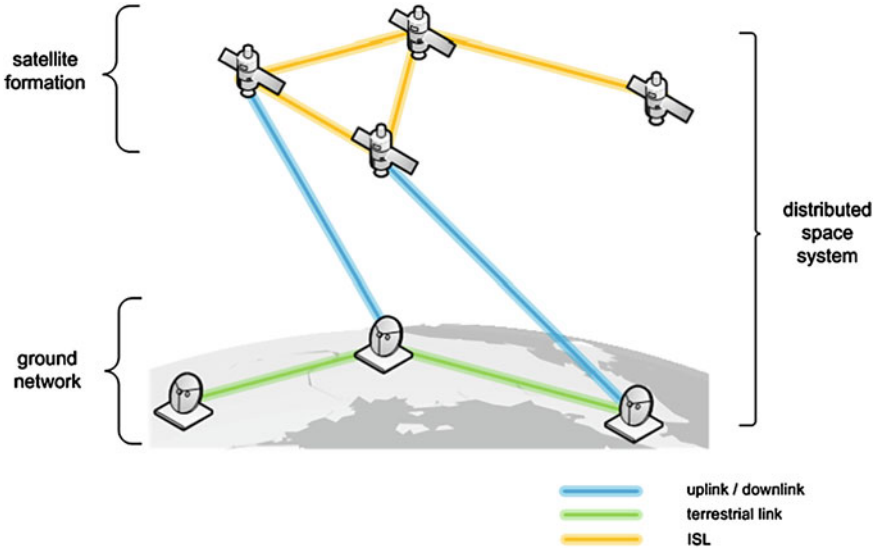

Fig. 2.2 Satellite constellation mission [37]

the satellites in their orbits, the communication links between ground stations and satellites change frequently and data flow in the satellite network has to be coordinated. A considerable challenge is also the invariance of the constellation geometry when subject to orbital perturbations (there could be a large number of possible constellation configurations that may satisfy a particular mission requirement).

Considering these issues, we determine the following *autonomy requirements* (also add the requirements of Polar LEO/Remote-Sensing Satellite Missions):

- *self-* requirements* (autonomicity):
  - *self-formation* (autonomously determine the right satellite configuration and perform it);
  - *self-reparation* (broken communication links must be restored autonomously);
  - *self-scheduling* (autonomously determine which satellites operate when; payload);
  - *self-coordination* (autonomously coordinate operations where several spacecraft may coordinate their operations on achieving a common goal: (1) Earth observation of a specific region performed by several spacecraft: at different times or with different instruments for sensor-fusion purposes and (2) coordination of scientific experiments);
  - *self-organization* (to distribute data in a space network a high degree of self-organization is required, i.e., autonomous routing capabilities due to changing topology);
  - *self-geometry* (autonomously adapt the constellation geometry to orbit perturbations);

- *knowledge*: constellation satellites (or neighboring satellites), inter-satellite communication links, group payload, constellation orbit planes, constellation geometry (e.g., Walker Delta pattern constellation), total number of satellites;
- *awareness*: formation awareness, satellites synchronization awareness;
- *monitoring*: constellation configuration;
- *adaptability*: adapt to new formations, adapt to weak inter-satellite communication links, adapt constellation geometry to orbital perturbations;
- *dynamicity*: dynamic formation (dynamic topology), dynamic inter-satellites communication links (change affects communication);
- *robustness*: robust to inter-satellite communication losses, robust to a single satellite loss;
- *resilience*: resilient satellite formations;
- *mobility*: inter-constellation mobility (information and satellites), moving satellites within an orbit plane, moving satellites from one orbit plane to another.

2.3.2.3 Geostationary Earth Orbit Missions

This class of missions involves satellites orbiting at Geostationary Earth Orbit (GEO) usually for providing global communications [20]. Satellites in such an orbit have an orbital period equal to one sidereal day (the Earth's rotational period or 23 h 56 min). The 24-h geostationary orbit clearly offers unique advantages, providing almost complete global coverage from merely three satellites, and with no need for the ground antenna to switch between the satellites. Several transfer orbit revolutions occur before injection of the satellite into near-circular, near-GEO orbit.

Mission Challenges and Generic Autonomy Requirements

A geostationary orbit can only be achieved at an altitude very close to 35,786 km (22,236 mi), and directly above the Equator.

Considering these issues, we determine the following *autonomy requirements* (also add the requirements of Polar LEO/Remote-Sensing Satellite Missions):

- *self-* requirements* (autonomicity):
  - *self-GEO-keeping* (use thrusters to autonomously maintain the geostationary orbit—position, altitude and speed, by adapting to perturbations such as the solar wind, radiation pressure, variations in the Earth's gravitational field, and the gravitational effect of the Moon and Sun);
- *knowledge*: GEO coordinates, perturbation factors, GEO altitude, solar wind, Moon's gravitational field, Sun's gravitational field;
- *awareness*: orbit perturbation awareness, solar wind awareness, radiation pressure awareness, Moon's gravitational effect awareness, Sun's gravitational effect awareness;
- *monitoring*: GEO position, other GEO satellites, Moon position, Sun position;

- *adaptability*: adapt to communication latency (geostationary orbits are far enough away from Earth that communication latency becomes significant—about a quarter of a second), adapt to perturbations (such as the solar wind, radiation pressure, variations in the Earth's gravitational field, and the gravitational effect of the Moon and Sun);
- *dynamicity*: dynamic GEO positioning and altitude;
- *robustness*: robust to communication latency;
- *resilience*: resilient GEO positioning;
- *mobility*: moving the satellite within the GEO plane.

2.3.2.4 Highly Elliptic Orbit Missions

In this class of missions, spacecraft in elliptic orbits move more rapidly at perigee than at apogee. This offers the prospect of a pass of increased duration over a ground station if the apogee is situated above it. Two mission subclasses are derived [20]: *Space-born Observatories* and *Communication Spacecraft*.

Space-borne Observatories

Spacecraft are used in observatory mode, which means the spacecraft instruments are operated as if they were located in a room adjacent to the astronomer's workstation. To achieve extended periods of time, the payload can be pointed to desired astrophysical targets whilst uninterrupted contact with a ground station is maintained. In general, there will be an interruption of observational time while the spacecraft passes through the perigee region.

Mission Challenges and Generic Autonomy Requirements

There are a few important challenges to consider in Space-borne Observatories. The first one is the orbit optimization, i.e., the spacecraft's orbit period must be optimized with respect to the ground station coverage. In addition, the radiation environment may preclude the operation of certain types of payload.

Considering these issues, we determine the following *autonomy requirements* (also add the requirements of Polar LEO/Remote-Sensing Satellite Missions):

- *self-* requirements* (autonomicity):
  - *self-optimization* (autonomously maintain the optimum spacecraft's orbit period with respect to the ground station coverage and keep up with it);
  - *self-protection* (autonomously detect high radiation and cover sensitive instruments);
  - *self-reparation* (autonomously detect problems in instruments and repair; broken communication links must be restored autonomously);
  - *self-command* (autonomously evaluate the effect of executing remote commands before perform those to guarantee that the spacecraft will not fall in a dangerous situation due to a command execution);
  - *self-scheduling* (autonomously determine which instruments operate when);

  - *self-coordination* (autonomously coordinate data flow gathered by different instruments onboard);
  - *self-tuning* (autonomously tune the instruments onboard);

- *knowledge*: instruments onboard, inter-instrument communication links, objects/phenomena to observe, Moon's gravitational field, Sun's gravitational field;
- *awareness*: operation awareness, instruments synchronization awareness, Moon's gravitational force awareness, Sun's gravitational force awareness;
- *monitoring*: instruments operation, Moon position, Sun position;
- *adaptability*: adapt to new tasks, adapt to instrument losses, adapt to instrument performance degradation;
- *dynamicity*: dynamic instrument configuration and tuning;
- *robustness*: robust to inter-instrument communication losses, robust to a single instrument loss;
- *resilience*: resilient instruments: (1) implies possible mitigations for the performance degradation and (2) autonomous recalibration to maintain the measurement data quality;
- *mobility*: inter-instrument mobility of information, moving observatory within an orbit plane, moving observatory from one orbit plane to another.

Communication Spacecraft

In this mission subclass, orbiting communication spacecraft fly highly ecliptic orbits and are used to transfer data on Earth. With regard to the ecliptic orbit, there two possible orbits [20]:

- *Molniya Orbit*—highly elliptic with a 12-h period where the spacecraft moves relatively slowly in the apogee region; to provide 24-h regional service, at least three Molniya spacecraft are needed.
- *Tundra Orbit*—elliptical orbit with a period one sidereal day (23 h 56 min). It can provide 24 h coverage with a minimum of only two spacecraft; the orbital parameters can be chosen so that the spacecraft does not traverse the Earth's radiation belts.

Mission Challenges and Generic Autonomy Requirements

There are a few important challenges to consider in Communication Spacecraft. The first one is the orbit perturbations, i.e., third-body forces may perturb the perigee height, causing atmosphere reentry. In addition, the radiation environment, e.g., a passage through Van Allen radiation belts, may cause accelerated degradation of power and electronic systems. Another challange is stemming from the variation in satellite range and range-rate, which may have a number of impacts upon the communication payload design:

- variation in time propagation;
- frequency variation due to Doppler effect;
- variation in received signal power;

- change of ground coverage pattern during each orbit.

Considering these issues, we determine the following *autonomy requirements* (also add the requirements of Polar LEO/Remote-Sensing Satellite Missions):

- *self-* requirements* (autonomicity):
  - *self-protection* (autonomously detect when the spacecraft is passing through the Van Allen radiation belts to cover the electronic systems and minimize the power usage);
  - *self-optimization* (autonomously optimize the communication payload by taking into consideration the impact caused by: variation in time propagation, frequency variation due to Doppler effect, variation in received signal power, and change of ground coverage pattern during each orbit);
  - *self-reparation* (autonomously detect problems in the communication system and repair);
  - *self-scheduling* (autonomously determine when to emit transmissions);
- *knowledge*: Van Allen radiation belts, Doppler effect, ground coverage pattern, Moon gravity, Sun gravity, Molniya Orbit/Tundra Orbit;
- *awareness*: signal power awareness, Moon's gravitational force awareness, Sun's gravitational force awareness;
- *monitoring*: Van Allen radiation belts, Moon position, Sun position;
- *adaptability*: adapt to changes in the ground coverage pattern, adapt to changes in time propagation, adapt to changes in communication frequency;
- *dynamicity*: dynamic communication frequency, dynamic ground coverage pattern, avoid radiation belts;
- *robustness*: robust to radiation;
- *resilience*: resilient communication payload;
- *mobility*: moving satellite within the orbit plane.

2.3.3 Interplanetary Missions

Interplanetary missions involve more than one planet or planet satellite. General trajectory information needs to be developed and understood for each mission. Interplanetary trajectories are influenced by perturbations caused by the gravitational influence of the Sun and planetary bodies within the solar system. Software tools are used to compute a large number of trajectories. Figure 2.3 presents possible trajectories for current Mars missions' opportunities [21].

Mission Challenges and Generic Autonomy Requirements

The major challenges to consider in Interplanetary Missions are communication propagation delays, communication bandwidth limitations, and hazardous environment. Communication propagation delays imply little or no possibility for real-time control by the ground station on Earth. The hazardous environment introduces risks such as surface interactions, thermal conditions, power availability, and radiation.

Considering these issues, we determine the following *autonomy requirements*:

- *self-* requirements* (autonomicity):
  - *self-trajectory* (autonomously acquire the most optimal trajectory; adapt to trajectory perturbations);
  - *self-protection* (autonomously detect the presence of radiation);
  - *self-scheduling* (autonomously determine what task to perform next—equipment onboard should support the tasks execution);
  - *self-reparation* (broken communication links must be restored autonomously; when malfunctioning, component should be fixed autonomously where possible);
- *knowledge*: mission objectives, payload operational requirements, instruments onboard together with their characteristics (e.g., instruments resolution), Van Allen radiation belt, ground stations, communication links, data transmission format,

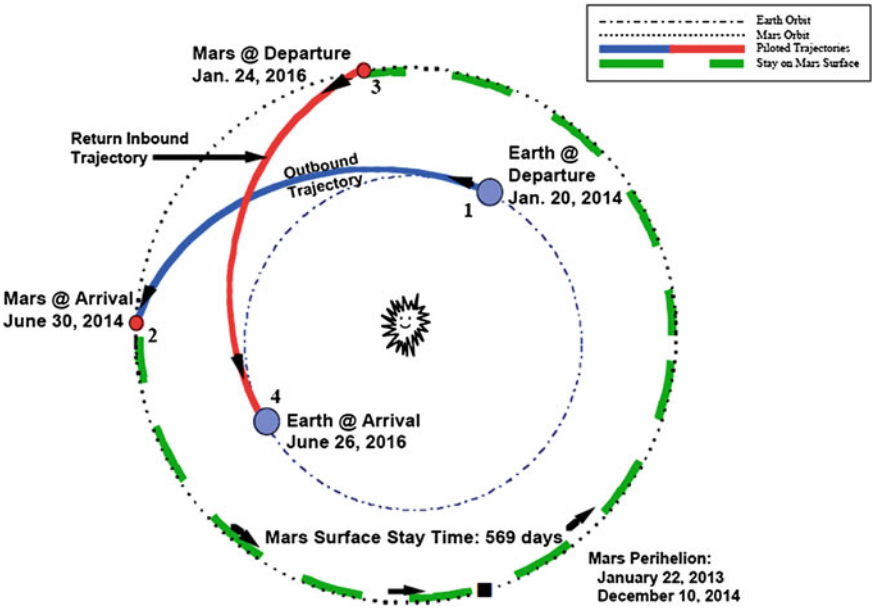

Fig. 2.3 Opportunities for Mars interplanetary missions, figure courtesy of George and Kos [21]

eclipse period, altitude, communication mechanisms onboard, Earth gravity, Moon gravity, Sun gravity, solar system, target planet characteristics;
- *awareness*: trajectory awareness, radiation awareness, air resistance awareness, instrument awareness, sensitive to thermal stimuli, gravitational forces awareness, data-transfer awareness, speed awareness, communication awareness;
- *monitoring*: electronic components onboard, surrounding environment (e.g., radiation level, space objects), planned operations (status, progress, feasibility, etc.);
- *adaptability*: adaptable mission parameters, possibility for re-planning (adaptation) of operations, adapt to loss of energy, adapt to high radiation, adapt to weak a satellite-ground station communication link, adapt to low energy;
- *dynamicity*: dynamic communication links;
- *robustness*: robust to temperature changes, robust to trajectory perturbations, robust to communication losses;
- *resilience*: loss of energy is recoverable, resilient to radiation;
- *mobility*: information goes in and out, changing trajectory;

2.3.3.1 Small Object Missions: "To Orbit" and "To Land" Missions

The objective of such missions is to investigate the properties of minor bodies in the solar system, mainly asteroids, comets and the satellites of the major planets [20]. Particular interest lies in the hypothesis that these small objects might help us understand the genesis and evolution of the solar system.

Mission Challenges and Generic Autonomy Requirements

Asteroids and cometary bodies are often characterized by an irregular shape, which poses some challenges to the mission designers:

- *monitor the environment around small objects*—often not known a priori and self-adaptation must be considered where a range of environments need to be accommodated in the mission design:
  - the near-body environment might not be clean, i.e., presence of dust and debris; a near-body environment exhibiting dust and debris will cause orbit and altitude perturbations;
  - cometary bodies can have a very dynamic near-body environment, particularly when closer than 3 AU to the Sun; comets may exhibit visible coma, hydrogen cloud and dust and plasma tails;
  - the gravity field does not approximate to that of a sphere;
- *motion around small, irregularly shaped bodies*:
  - find the sufficient distance to approximate to the Keplerian Orbits; for close orbits where the orbit radius is of the same order as the body's size the trajectory shape no longer approximates to a conic section;
  - determine the gravitational field of the body; payload imagery can provide quantitative information regarding the size, shape and rotational state of the body;

  - determine the overall mass and density of the body—can use the so-called "mascon" method that builds the body's shape via a collection of spherical masses of uniform size and density;

- *landing on small, irregularly shaped bodies*:
  - physical properties of the comet and its environment are generally unknown prior to the mission; we need to build generic models of the target body (e.g., a comet), which encompass a range of properties expected; key factors to determine: orbital speed, orbital period, escape velocity and impact speed;
  - landing takes long time and requires trajectory manoeuvers and it is important not to exceed escape velocity; the lander needs to have knowledge and control of its altitude.

Considering these issues, we determine the following *autonomy requirements* (also add the generic requirements for Interplanetary Missions):

- *self-* requirements* (autonomicity):
  - *self-orbiting* (autonomously acquire the most optimal orbit; adapt to orbit perturbations due to possible dust and derbies);
  - *self-protection* (autonomously detect the presence of dust and derbies and move to escape or cover instruments);
  - *self-landing* (autonomously adapt the landing procedure to landing goals);
  - *self-gravity* (autonomously compute the gravitational field of the object—mass, density and shape of the object);
  - *self-escape* (autonomously acquire the escape procedures and use it to leave the object);

- *knowledge*: physics of minor space bodies, physics of cometary bodies, Keplerian Orbits;
- *awareness*: awareness of dust and debris in the object surrounding environment, object shape awareness, object gravitational force, object spin awareness, object speed awareness, surface distance awareness, spacecraft's orbital speed awareness, spacecraft's orbital period awareness, escape velocity awareness, body's magnetic fields awareness;
- *monitoring*: the environment around the target object, the landing surface, comet's characteristics (such as: visible coma, hydrogen cloud and dust and plasma tails);
- *adaptability*: adapt the system operations to the goals to be achieved, adapt to the environment around the target object (landing and orbiting operations must take into consideration both orbit and altitude perturbations), adapt to the shape of the orbited object, adapt to the landing surface;
- *dynamicity*: dynamic near-body environment, dynamic landing procedure (may require trajectory maneuvers);
- *robustness*: robust to dust and debris;
- *resilience*: resilient to magnetic fields changes;
- *mobility*: trajectory maneuvers for landing and orbiting.

2.3.3.2 Missions Using Low-Thrust Trajectories

Such missions use spacecraft for orbit control activities in GEO (Geostationary Earth Orbit), drag compensation in LEO, Lunar orbit missions and missions to comets and asteroids. These missions often have a complex mission profile utilizing ion propulsion in combination with multiple *gravity-assist manoeuvers* (e.g., ESA's BepiColombo mission).

Mission Challenges and Generic Autonomy Requirements

Major challenges to consider in Missions Using Low-thrust Trajectories are:

- low-thrust trajectories in a central force field—the motion of an orbiting spacecraft is influenced by a continuous low-thrust: a challenging task is to determine a steering law for the thrust vector, so that a particular objective can be achieved; direct trajectory cannot be implemented using such low thrust, and so a spiraling transfer trajectory must be used.
- steering laws: secular rates of the orbit elements, maximum rate of change of orbital energy, maximum rate of change of orbital inclination;
- interplanetary missions using low-thrust:
  - low-thrust Earth escape;
  - low-thrust planetary capture—achieve capture around the destination planet;
  - sun-centered transfer;

Considering these issues, we determine the following *autonomy requirements* (also add the generic requirements for Interplanetary Missions):

- *self-* requirements* (autonomicity):
  - *self-low-thrust-trajectory* (autonomously determine a steering law for a thrust vector);
  - *self-capture* (autonomously determine a steering law and use low thrust to achieve capture around a destination planet);
  - *self-escape* (autonomously determine a steering law and use low thrust to achieve escape from a planet);
- *knowledge*: central force field physics, steering law models, secular rates of the orbit elements, maximum rate of change of orbital energy, maximum rate of change of orbital inclination;
- *awareness*: planetary capture awareness, planetary escape awareness, trajectory velocity awareness, planet's magnetic fields awareness, awareness of the spacecraft's position on the projected trajectory;
- *monitoring*: the environment around the planet;
- *adaptability*: adapt the low thrust trajectory to orbit and/or altitude perturbations;
- *dynamicity*: dynamic near-body environment, dynamic trajectory following procedure (may require trajectory maneuvers);
- *robustness*: robust to dust and debris;

- *resilience*: resilient to magnetic fields changes;
- *mobility*: trajectory maneuvers for avoiding orbit and/or altitude perturbations.

2.3.3.3 Planetary Atmospheric Entry and Aeromaneuvering Missions

Such missions require entering the atmosphere of a planet to take probes or land [20]. Principle effect of an atmosphere on a satellite trajectory is to reduce the energy of the orbit. These missions include some degree of aeromaneuvering, with a "mass penalty"—additional propellant mass is required to protect the vehicle from the dynamic pressure and thermal effects of aeromaneuvering.

Mission Challenges and Generic Autonomy Requirements

Major challenges to consider in Missions Using Low-thrust Trajectories are:

- along-track effects: associated with aerobraking and reduce the translational energy of a spacecraft:
  - direct atmospheric entry—aeroforces are used to reduce the vehicle's speed to facilitate soft landing;
  - orbital aerocapture—aeroforces are used to transfer a vehicle's orbital state from hyperbolic to elliptical;
  - aero-assisted orbit transfer—an atmospheric pass is used to modify the orbit (e.g., transferring a vehicle from high orbit to low orbit by using an aerobraking maneuver);
- across-track effects: produce out-of-plan accelerations and can be used to modify the orbit plane inclination;
- atmospheric entry: a space vehicle experiences approximate exponentially increasing atmospheric density; provides a changing aerodynamic environment. As the vehicle progresses into the atmosphere tick shock waves are formed about the vehicle. Eventually, if the vehicle penetrates sufficiently low into the atmosphere, a continuum flow region is encountered where velocity remains hypersonic. Particular challenges here are:
  - the process is difficult to describe analytically and often bridging functions are used to describe the aerodynamic properties of the vehicle;
  - high temperature is generated at the contact surface that must be absorbed by the heat shield and vehicle surface. The high temperature leads to chemical reactions in the atmospheric gas and excitation of internal energy models such as vibration, together dissociation and ionization;
- principle constraints: peak dynamic load and peak thermal load, together with how long these loads persist;

Considering these issues, we determine the following *autonomy requirements* (also add the autonomy requirements for Small Object Missions—"to orbit" and "to land" Missions):

- *self-* requirements* (autonomicity):
  - *self-orbital-aerocapture* (autonomously use the aeroforces to transfer a vehicle's orbital state from hyperbolic to elliptical);
  - *self-aero-assisted-orbit-transfer* (autonomously use atmospheric passes to modify the orbit);
  - *self-soft-landing* (autonomously use aeroforces to reduce the vehicle's speed and perform soft landing);
  - *self-orbit-inclination* (autonomously use out-of-plan accelerations to modify the orbit plane inclination);
- *knowledge*: aerodynamic properties of the spacecraft, spacecraft's heat shield, across-track effects, atmospheric shock waves, gas chemical reactions, peak dynamic load, peak thermal load;
- *awareness*: atmospheric density awareness, temperature awareness, across-track effect awareness, shock waves awareness, vibration awareness, gas dissociation awareness, gas ionization awareness, chemical reactions awareness;
- *monitoring*: atmospheric density, temperature at the contact surface of the spacecraft, atmospheric chemical reactions;
- *adaptability*: adapt to out-of-plan accelerations;
- *dynamicity*: changing aerodynamic environment, dynamic velocity (depends on the atmospheric density);
- *robustness*: robust to high temperatures and steering forces;
- *resilience*: resilient to changes in the atmospheric density;
- *mobility*: trajectory aeromaneuvering for landing and orbiting.

2.4 Controller Architectures for Robotic Systems

In this section we elaborate on the so-called controller architectures for robotic systems to conclude the possible platforms for autonomous behavior controllers. This will add on to the generic autonomy requirements for space missions and will help us to conclude the characteristics of possible platforms (e.g., formal methods) for ARE.

2.4.1 Architectural Issues Related to Autonomy

By introducing (or increasing) autonomy to space missions, we imply changes to both the *control architecture* and the *commanding protocol*. This is mainly due to the fact that most of the operational conditions are not fully known in advance and we often need to deal with parameterized behaviors. Thus, we do not precisely know at design time the sequence of low-level commands that will be executed when a high-level command is issued. In situations when the system operates with performance

and resource uncertainties as well as other constraints stemming from the partially unknown environment, "the use of event-driven sequence execution is a much better approach than the traditional time-triggered command execution because commands executed at a precise time will fail under unexpected circumstances" [32]. Recall that systems following event-driven sequence execution are classified as having E3 level of autonomy according to ESA's autonomy classification (Sect. 2.1.2). The occurrence of events is non-deterministic and so the sequence of low-level actions that are executed by the system (and even the result). Actually, this is what makes autonomous systems difficult to verify and test.

The highest level of autonomy is goal-oriented operation (E4 level) (Sect. 2.1.2). From the autonomy-design and -implementation perspective, this level provides for the highest level of abstraction because goals have the advantage of *being easier to specify than the actions that are necessary to reach them* and the responsibility of deciding how to best achieve these goals is transferred to the *onboard controllers*. This level of autonomy has been tested successfully in missions such as Deep Space 1 [30] and Earth Observing Mission-1 (EO-1) [31] and is based on *automated reasoning*.

2.4.2 Controller Architectures for Robotic Systems

Architectures for *autonomous control* in robotic systems require concurrent embedded real-time performance, and are typically too complex to be developed and operated using conventional programming techniques. The complexity demands of such systems require frameworks and tools that are based on well-defined concepts that enable the effective realization of systems to meet high-level goals. Experience shows that layering is a powerful means for structuring functionalities and control, being therefore a powerful tool for system design, increasing modularity, robustness, ease of debugging, computational capability, reactivity and use of information to make decisions, therefore the need of these systems to contain several sub-systems with different levels of abstraction.

The core of an autonomous controller is a subsystem, known variously as the *execution system*, *virtual machine*, or *sequence engine*, that executes commands and monitors the environment [32]. Execution systems vary in sophistication, from those that execute linear sequences of commands at fixed times, to those that can plan and schedule in reaction to unexpected changes in the environment. Every robotic system requires some sort of execution system, although the level of autonomy and complexity of the controller varies greatly.

Three main architecture classes for autonomous control in robotic systems are in use: *deliberative architectures*, *reactive architectures*, and *hybrid architectures*.

2.4.2.1 Deliberative Architectures

The so-called *deliberative architecture controllers* are based on a particular type of *knowledge-based system* that contains an explicitly represented *symbolic model of the world*. Deliberation is the explicit consideration of alternative behaviors (courses of actions)—i.e., generating alternatives and choosing one of the possible alternatives. Decisions are made via *logical and statistical reasoning* based on pattern matching and symbolic manipulation. The approach suggests that intelligent behavior can be generated by providing a system with a symbolic representation of its environment and its desired behavior and by syntactically manipulating this representation and the decision-making is viewed as a logic deduction. Deliberative architectures are goal-oriented and thus, they are suitable for building the highest possible level of autonomy (E4 level).

2.4.2.2 Reactive Architectures

The so-called *reactive architecture controllers* differ from deliberative controllers in that they have a very limited set of beliefs and instead of *explicitly defined goals*; their actions are determined by a set of behaviors which are triggered by events within the environment. In reactive architectures, there are no central functional modules, such as perception reasoning, learning, etc. Instead an agent consists of a completely distributed decentralized set of competence modules, more often called behaviors. The best known reactive architecture is the so-called *Subsumption Architecture*. It is created to expose a number of behaviors (each behavior may be thought of as an individual action function) which are implemented as Finite State Machines (FSM). Reactive architectures are event-oriented and are suitable for building the E3 level of autonomy.

2.4.2.3 Hybrid (Layered) Architecture

The so-called *layered architectures* put together both deliberative and reactive architectures by breaking down the different proactive and reactive elements of an agent into different layers. This way of abstraction allows complex agents to be modeled more easily and is flexible enough for use in many agent systems. Each function required of the agent is decomposed into a different layer. The reactive components are responsible for relatively simple, low-level, robust behaviors, while the deliberative components are responsible for organizing and sequencing complex behaviors. The key issue is in the *integration of the reactive and deliberative layers*. The favored option is to use a *three-tiered approach* in which a planning component mediates between low-level behaviors and high-level goal-oriented modeling [32].

Figure 2.4 presents a three-tiered architecture. As shown there are three main layers [32]:

- a Decision Layer is a mechanism for performing time-consuming deliberative computations, with goal-driven planning and scheduling facilities;
- an Executive Layer is a reactive plan execution mechanism, with execution sequencing facilities;
- a Control (Functional) Layer is a reactive feedback control mechanism, with reactive execution facilities.

In three-layer architectures the three components run as separate computational processes. Reactive layers typically use very simple representations of the current or previous state of the environment and work over very short timescales in tight sensor-driven feedback loops, while deliberative layers use complex counterfactual representations and work on much longer timescales, from minutes to hours and more [32].

In regard to layer dependency and positioning, there are a variety of hybrid architectures, some of which are presented below.

Horizontal and Vertical Layering

In horizontal layering the controller consists of behaviors that take input from the environment and create some sort of output. In vertical layering the environmental input triggers a low layer which then passes information to the layer above it and so on allowing for considerable complexity. The great advantage of horizontally layered architectures is their conceptual simplicity: if we need a controller to exhibit *different types of behavior we implement* different layers.

One-pass and Two-pass Vertical Layered Architectures

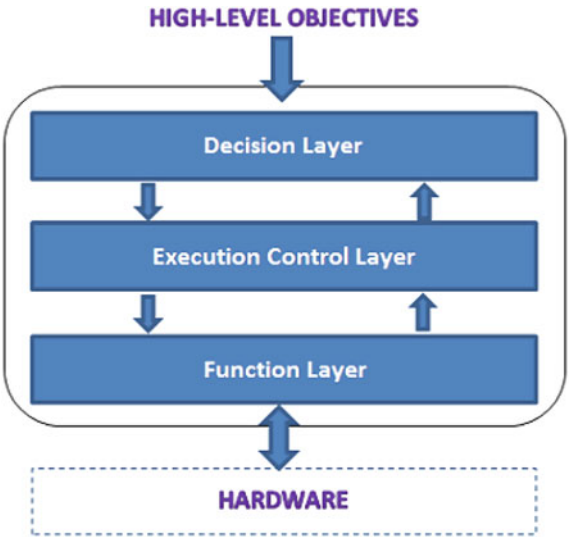

Fig. 2.4 Three-tiered hybrid controller architecture [32]

In the first, control flows sequentially through each layer, until the final layer generates an action output. In the second, information flows up the architecture and control flows back down. In decentralized control, the layers operate concurrently and independently, processing sensor data and generating actions. In hierarchical control the layers operate serially with higher level deliberative layers controlling the execution of low-level reactive layers. In concurrent control the layers operate simultaneously and can modify the behavior of adjacent layers.

2.5 Formal Methods for ARE

Formal methods originate from formal logic and refer to a plethora of mathematically-based activities where both notations and tools with mathematical basis are used. Formal notations (also known as formal languages) are used to precisely and unambiguously specify system requirements or to model system design. Formal tools provide the needed formal support to prove correctness of specified system properties and eventually correctness of their implementation.

Formal methods provide a vast family of formal notations in the form of formal specification languages (sometimes called modeling languages). In general, we use these to precisely describe with the logical underpinning of mathematics features of the system under consideration at a higher level of abstraction than the one provided by implementation. Thus, formal languages are used to provide a formal and neutral, i.e., implementation-independent, representation (often called formal specification) of systems. In avionics, formal methods are used for requirements specification, system modeling and code generation, and validation.

Formal languages have been used to develop deliberative robot controllers (Sect. 2.4.2). In [60] an agent programming language called Goal is used to program a cognitive robot control architecture that combines low-level sub-symbolic control with high-level symbolic control. The Goal language helps to realize a cognitive layer whereas low-level execution control and processing of sensor data are delegated to components in other layers. Goal supports a goal-oriented behavior and decomposition of complex behavior by means of modules that can focus their attention on relevant sub-goals. In [39] the high-level language Golog is used for robot programming. Golog supports writing control programs in a high-level logical language, and provides an interpreter that, given a logical axiomatization of a domain, will determine a plan. Golog also supports actions and situations (actually the language incorporates Situation Calculus [7]).

Autonomic System Specification Language (ASSL) is a declarative specification language for ASs with well-defined semantics [54–56]. It implements modern programming language concepts and constructs like inheritance, modularity, type system, and high abstract expressiveness. Conceptually, ASSL is defined through formalization tiers. Over these tiers, ASSL provides a multi-tier specification model that is designed to be scalable and exposes a judicious selection and configuration of

infrastructure elements and mechanisms needed by an AS. ASSL has been successfully used for the specification of autonomic features of a variety of ASs.

KnowLang [49, 50] is an approach to knowledge representation model for self-adaptive behavior and awareness. The ultimate goal is to allow for awareness and self-awareness capabilities (Sect. 2.5.2) of autonomous systems. The KnowLang framework strives to solve complex problems where the operational environment is non-deterministic and a system needs to reason at runtime to find missing answers. In comparison to Goal and Golog described above, KnowLang is far more expressive especially at the level of modeling self-adaptive behavior, which is supported neither by Goal nor by Golog. KnowLang supports integration of situations, goals, policies, and actions with Bayesian network probability distributions which allows for self-adaptation based on both *logical* and *statistical reasoning*.

In the following subsections, we present in detail the goal-oriented requirements engineering approach along with two successful formal methods: ASSL and KnowLang, which we consider as the most promising approaches to ARE for space missions. Note that these methods cope well with both the generic autonomy requirements for space missions (Sect. 2.3) and controller architectures for robotic systems (Sect. 2.4).

2.5.1 Goal-Oriented Requirements Engineering

The *goal-oriented approach* to Software Requirements Engineering is about capturing and analyzing stakeholder intentions to derive functional and non-functional (hereafter quality) requirements [12, 29]. In essence, this approach extends upstream the software development process by adding a new phase called *Early Requirements Analysis*. The fundamental concepts used to drive the goal-oriented form of analysis are those of *goal* and *actor*. To fulfill a stakeholder goal, the Goal-Oriented Requirements Engineering (GORE) [46] approach provides for *analyzing the space of alternatives*, For more details on GORE, please, refer to Sect. 1.5.1 in Chap. 1

2.5.1.1 GORE for Autonomy Requirements

There has been interest in trying to apply GORE to tackle requirements for ASs [27]. The basic idea is to build *goal models* that can help us to consecutively design autonomous systems in several ways:

1. A goal model might provide the starting point for the development of an AS by analyzing the environment for the system-to-be and by identifying the problems that exist in this environment as well as the needs that the system under development has to address:

(a) GORE might assist the Space Mission Requirements Analysis Process (Sect. 2.3.1) in defining mission objectives and constraints and in estimating quantitative mission needs and requirements.
(b) With GORE mission goals can be identified along with the mission actors (mission spacecraft, spacecraft components, environmental elements, base station, etc.).
(c) Requirements goal models can be used as a *baseline for validating the system*.

2. Goal models provide a means to represent *alternative ways* in which the objectives of the system can be met and analyze and rank these alternatives with respect to *quality concerns* and other constraints:

(a) This allows for exploration and analysis of alternative system behaviors at design time, which leads to more predictable and trusted ASs.
(b) If the alternatives that are initially delivered with the system perform well, there is no need for complex interactions on autonomy behavior among autonomy components.
(c) Of course, not all alternatives can be identified at design time. In an open and dynamic environment, new and better alternatives may present themselves and some of the identified and implemented alternatives may become impractical.
d) In certain situations, new alternatives will have to be discovered and implemented by the system at runtime. However, the process of discovery, analysis, and implementation of new alternatives at runtime is complex and error-prone. By exploring the space of alternative process specifications at design time, we are minimizing the need for that difficult task.

3. Goal models provide the traceability mechanism from AS designs to requirements. When a change in requirements is detected at runtime (e.g., a major change in the global mission goal), goal models can be used to re-evaluate the system behavior alternatives with respect to the new requirements and to determine if system reconfiguration is needed:

(a) If a change in requirements affected a particular goal in the model, it is possible to see how this goal is decomposed and which autonomy components (autonomic elements) implementing the goal are in turn affected.
(b) By analyzing the goal model, it is possible to identify how a failure to achieve some particular goal affects the overall objective of the system.
(c) Highly variable goal models can be used to visualize the currently selected system configuration along with its alternatives and to communicate suggested configuration changes to users in high-level terms.

4. Goal models provide a unifying intentional view of the system by relating goals assigned to individual autonomy components (autonomic elements) to high-level system objectives and quality concerns:

(a) High-level objectives or quality concerns serve as the common knowledge shared among the autonomy components to achieve the global system optimization. This way, the system can avoid the pitfalls of missing the globally optimal configuration due to only relying on local optimizations.
(b) Goal models might be used to identify part of the knowledge requirements of the system (Sects. 2.2.3 and 2.3).

5. GORE might be used to manage conflicts among multiple goals including self-* objectives (autonomicity requirements) (Sects. 2.2.3 and 2.3). Goals have been recognized to provide the roots for detecting conflicts among requirements and for resolving them eventually [34, 45]. Note that by resolving conflicts among goals or obstacles to goal achievement, new goals (or self-* objectives) may emerge.
6. Resilience and robustness autonomy requirements might be handled by GORE as soft-constraints. For example, such requirements for GEO Missions (Sect. 2.3.2.3) are defined as *robustness: robust to communication latency* and *resilience: resilient GEO positioning*. These requirements can be specified as soft-goals leading the system towards *reducing and copying with communication latency* and *keeping GEO positioning optimal*. Note that specifying soft goals is not an easy task. The problem is that there is no clear-cut satisfaction condition for a soft-goal. Soft-goals are related to the notion of satisfaction [38]. Unlike regular goals, soft-goals can seldom be accomplished or satisfied. For soft-goals, eventually, we need to find solutions that are "*good enough*" where soft-goals are satisficed to a sufficient degree. Thus, when specifying robustness and resilience autonomy requirements we need to set the desired degree of satisfaction, e.g., by using probabilities.
7. Monitoring, mobility, dynamicity and adaptability might also be specified as soft-goals, but with relatively high degree of satisfaction. These three types of autonomy requirements represent important quality requirements that the system in question need to meet to provide conditions making autonomicity possible. Thus, their degree of satisfaction should be relatively high. Eventually, adaptability requirements might be treated as hard goals because they determine what parts of the system in question can be adapted (not how).

2.5.1.2 Goal Modeling

The benefit of goal modeling is to support heuristic, qualitative or formal reasoning schemes during requirements engineering. Goals are generally modelled by intrinsic features such as their type and attributes, and by their links to other goals and to other elements of a requirements model. Goals can be hierarchically organized and prioritized where high-level goals (e.g., mission objectives) might comprise related, low-level, sub-goals that can be organized to provide different alternatives of achieving the high-level goals. For example, KnowLang specifies goals either as a *transition from a state to a desired state* or simply as *a transition to a desired*

state (Sect. 2.5.4.2). In this approach, a state presents particular conditions that must be met involving the system, environment or both, and a goal can be achieved via multiple inter-state-transitions with alternatives.

Goals are modeled with ASSL as *service-level objectives* (SLO) (Sect. 2.5.3) that can be correlated with events. SLO events can be fired when objectives get degraded or normalized, which allows the system react to changes in goals realization.

Goal modeling with languages like KnowLang, can be used to develop deliberative controllers for autonomous spacecraft. For example, KnowLang helps to realize a cognitive layer whereas system goals are pursued and low-level execution control and processing of sensor data are delegated to different components. Goal modeling supports a goal-oriented behavior and decomposition of complex behavior by means of policies that can focus on relevant sub-goals (Sect. 2.5.4.2).

2.5.2 Awareness Modeling

Awareness generally is classified into two major areas: *self-awareness*, pertaining to the internal world, and *context-awareness*, pertaining to the external world (Sect. 2.2.3). Autonomic computing research defines these two classes [25] as following:

- A self-aware system has detailed knowledge about its own entities, current states, capacity and capabilities, physical connections, and ownership relations with other systems in its environment.
- A context-aware system knows how to sense, negotiate, communicate, and interact with environmental systems and how to anticipate environmental system states, situations, and changes.

Perhaps a third class could be *situational awareness*, which is self-explanatory. Other classes could draw attention to specific problems, such as operational conditions and performance (operational awareness), control processes (control awareness), interaction processes (interaction awareness), and navigation processes (navigation awareness). Although classes of awareness can differ by subject, they all require a subjective perception of events and data "within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future" [15].

To better understand the idea of awareness in space missions, consider an exploration robot. Its navigation awareness mechanism could build a map on the fly, with landmarks represented as part of the environment knowledge, so that navigation becomes simply a matter of reading sensor data from cameras and plotting the robot's position at the time of observation. Via repeated position plots, the robot's course and land-reference speed can be established.

Recent research efforts have focused on awareness implementations in software-intensive systems. For example, commercially available server-monitoring platforms, such as Nimbus [44] and Cittio's Watch Tower [43], offer robust, lightweight

sensing and reporting capabilities across large server farms. Such solutions are oriented toward massive data collection and performance reporting, so they leave much of the final analysis and decision making to a human administrator. Other approaches achieve awareness through model-based detection and response based on offline training and models constructed to represent different scenarios that the system can recognize at runtime.

To function, the mechanism implementing the awareness must be structured to take into consideration different stages—for example, it might be built over a complex chain of functions such as *raw data gathering*, *data passing*, *filtering*, *conversion*, *assessment*, *projection*, and *learning* [58]. As Fig. 2.5 shows, ideally, all the awareness functions could be structured as an awareness pyramid, forming the mechanism that converts raw data into conclusions, problem prediction, and eventually learning. As shown in Fig. 2.5, the first three levels include monitoring tasks; the fourth, recognition tasks; the fifth and sixth, assessment tasks; and the last, learning tasks. The pyramid levels in Fig. 2.5 represent awareness functions that can be grouped into four specific tasks:

- *monitoring*—collects, aggregates, filters, manages, and reports internal and external details such as metrics and topologies gathered from the system's internal entities and its context;
- *recognition*—uses knowledge structures and data patterns to aggregate and convert raw data into knowledge symbols;
- *assessment*—tracks changes and determines points of interest, generates hypotheses about situations involving these points, and recognizes situational patterns;
- *learning*—generates new situational patterns and maintains a history of property changes.

Aggregation can be included as a subtask at any function level; it's intended to improve overall awareness performance. For example, it can pull together large amounts of sensory data during the filtering stage or recognition tasks can apply it to improve classification.

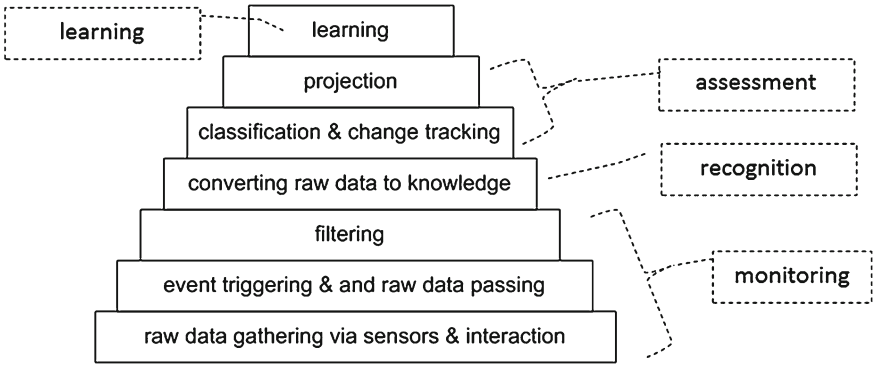

Fig. 2.5 The awareness pyramid [58]

The awareness process isn't as straightforward as it might seem- rather, it's cyclic, with several iterations over the various awareness functions. Closing the chain of awareness functions can form an awareness control loop in which different awareness classes can emerge [57]. The process's cyclic nature is why awareness itself is so complex, with several levels of exhibition and degrees of perception. The levels can be related to data readability and reliability—that is, they might include noisy data that must be cleaned up and eventually interpreted with some degree of probability. Other levels might include early awareness, which is a product of one or two passes of the awareness control loop, and late awareness, which should be more mature in terms of conclusions and projections.

2.5.3 ASSL

ASSL [54–56] is a model-oriented specification language for autonomic systems (ASs) (Chap. 1). ASSL considers ASs, as composed of *autonomic elements* (AEs) (Chap. 1) interacting over special *interaction protocols*.

To specify ASs, ASSL uses a multi-tier specification model. By their nature, the ASSL tiers are levels of abstraction of different aspects of the AS under consideration. These provide domain-specific constructs (specific to the autonomic computing domain) such as self-management policies, communication interfaces, execution semantics, actions, etc. There are three major tiers (three major abstraction perspectives), each composed of sub-tiers.

- *AS Tier*—forms a general and global AS perspective, where we define the general system rules in terms of service-level objectives (SLO) and self-management policies, architecture topology, and global actions, events, and metrics applied in these rules.
- *AS Interaction Protocol (ASIP) Tier*—forms a communication protocol perspective, where we define the means of communication between the so-called autonomic elements (system components).
- *AE Tier*—forms a unit-level perspective, where we define interacting sets of AEs with their own behavior.

Figure 2.6 represents the tiers in ASSL. As we can see, the ASSL specification model decomposes an AS in two directions: first into levels of functional abstraction, and second into functionally related sub-tiers. This decomposition fits naturally to the needs of the ASs builders, i.e., to build an AS:

1. We need a plan, i.e., a global picture of the entire system, which is specified at the AS tier.
2. We need the building blocks to construct that system—we specify our "bricks" at the AE tier.
3. We need the glue to put the bricks together and build the system. Our glue is the communication protocol, which we specify at the ASIP tier.

```
I. Autonomic System (AS)
  * AS Service-level Objectives
  * AS Self-managing Policies
  * AS Architecture
  * AS Actions
  * AS Events
  * AS Metrics
II. AS Interaction Protocol (ASIP)
  * AS Messages
  * AS Communication Channels
  * AS Communication Functions
III. Autonomic Element (AE)
  * AE Service-level Objectives
  * AE Self-managing Policies
  * AE Friends
  * AE Interaction Protocol (AEIP)
    - AE Messages
    - AE Communication Channels
    - AE Communication Functions
    - AE Managed Elements
  * AE Recovery Protocol
  * AE Behavior Models
  * AE Outcomes
  * AE Actions
  * AE Events
  * AE Metrics
```

Fig. 2.6 ASSL multi-tier specification model

ASSL is designed to tackle the so-called *self-* requirements* (Sect. 2.2.3). To do so, the framework imposers a requirements engineering approach where self-* requirements are specified as special *policy models* [54, 55]. Note that the ASSL policy models specify special self-* policies (e.g., self-healing) driving the system in critical situations. To specify ASSL policy models, we need to come up with self-adapting scenarios where we may consider important events, actions, SLOs, etc. Note that the ASSL SLOs are actually high-order goal models where the GORE approach (Sect. 2.5.1) can be successfully applied to elicit goals and connect those goals with the proper policies. ASSL has been used in a variety of projects targeting functional prototypes of autonomous NASA space exploration missions [48, 52].

2.5.3.1 Specifying and Generating Prototypes with ASSL

The ASSL tiers are intended to specify different aspects of the AS in question, but it is not necessary to employ all of those in order to model an AS. Usually, an ASSL specification is built around self-management policies, which make that specification AC-driven. The ASSL formal model addresses policy specification at both AS and AE tiers. Policies are specified with special constructs called *fluents* and *mappings*:

- *Fluents* are states with duration and when the system gets into a specific fluent, a policy may be activated.
- *Mappings* map particular fluents to particular actions to be undertaken by the specified AS.

ASSL expresses fluents with *fluent-activating* and *fluent-terminating events*, i.e., the self-management policies are driven by events. In order to express mappings, conditions and actions are considered, where the former determine the latter in a deterministic manner.

The following ASSL code presents an example specification of a self-healing policy. Please refer to [54, 55] for more details on the ASSL specification model and grammar.

```
ASSELF_MANAGEMENT {
 SELF_HEALING {
  FLUENT inLosingSpacecraft {
   INITIATED_BY { EVENTS.spaceCraftLost }
   TERMINATED_BY { EVENTS.earthNotified }
  }
  MAPPING {
   CONDITIONS { inLosingSpacecraft }
   DO_ACTIONS { ACTIONS.notifyEarth }
  }
 }
} // ASSELF_MANAGEMENT
```

Once a specification is complete, it can be validated with the ASSL built-in consistency checking mechanism and a functional prototype can be generated automatically. The prototypes generated with the ASSL framework are fully-operational multithreaded event-driven applications with embedded messaging.

2.5.3.2 Handling Event-Based Autonomy with ASSL

ASSL is designed to tackle the *self-* requirements* (Sect. 2.2.3). In addition, ASSL could be extremely powerful when handling the ESA's L3 event-based autonomy (Sect. 2.1.2). ASSL aims at event-driven autonomic behavior. Recall that to specify self-management policies, we need to specify appropriate events (Sect. 2.5.3.1). Here, we rely on the reach set of event types exposed by ASSL [54, 55]. For example, to specify ASSL events, one may use logical expressions over SLOs, or may relate events with metrics, other events, actions, time, and messages. Moreover, ASSL allows for the specification of special conditions that must be stated before an event is prompted. Therefore, events can be constrained by adding such conditions to their specification.

Finally, ASSL appears to be very well suited for developing autonomous controllers for reactive architectures (Sect. 2.4.2.2). It is event-driven (exposes a rich set of possible events raised in the environment and the system itself). Special competence models can be built by using the self-adaptive policy and relying on ASSL fluents and actions to provide for desired behaviors. Note that ASSL implies layering (Fig. 2.6) for structuring functionalities in event-driven autonomy and provides computational structures that can be possibly effective when handling *layered controller architectures* (Sect. 2.4.2.3).

2.5.4 KnowLang

KnowLang is a framework for Knowledge Representation and Reasoning (KR&R) that aims at efficient and comprehensive *knowledge structuring and awareness* based on logical and statistical reasoning. It helps software engineers to tackle requirements via (1) explicit representation of domain concepts and relationships; (2) explicit representation of particular and general factual knowledge, in terms of predicates, names, connectives, quantifiers and identity; and (3) uncertain knowledge in which additive probabilities are used to represent degrees of belief.

A key feature of KnowLang is a *multi-tier specification model* [49, 50] allowing for integration of ontologies together with rules and Bayesian networks. At its very core, KnowLang is a formal specification language providing a comprehensive specification model aiming at addressing the knowledge representation problem for intelligent systems including engineering autonomy requirements. The complexity of the problem necessitated the use of a specification model where knowledge (consider also knowledge requirements) can be presented at different levels of abstraction and grouped by following both hierarchical and functional patterns. KnowLang imposes a multi-tier specification model (Fig. 2.6) where we specify a special knowledge base (KB) composed of layers dedicated to knowledge corpuses, KB (knowledge base) operators and inference primitives [49, 50].

The tier of knowledge corpuses is used to specify KR structures. The tier of KB operators provides access to knowledge corpuses via special class of *ASK* and *TELL operators* where ASK operators are dedicated to knowledge querying and retrieval and TELL operators allow for knowledge update. Moreover, this tier provides for special inter-ontology operators intended to work on one or more ontologies specified within the knowledge corpuses. Note that all the KB operators may imply the use of *inference primitives*, i.e., new knowledge might be inferred and eventually stored in the KB. The tier of inference primitives is intended to specify algorithms for reasoning and knowledge inference.

For more details on the KnowLang's multi-tier specification model, please refer to [49, 50].

2.5.4.1 Requirements Engineering with KnowLang

KnowLang can be successfully used to capture both functional (behavior requirements) and non-functional requirements (data requirements, constraints, etc.). Exclusively dedicated to knowledge modeling, KnowLang provides a rich set of constructs that helps us specify data requirements and functional requirements following a goal-oriented and/or behavior-oriented approach. When we specify knowledge with KnowLang, we build a KB with a variety of knowledge structures such as *ontologies*, *facts*, *rules*, and *constraints* where we need to specify the ontologies first in order to provide the "*vocabulary*" for the other knowledge structures.

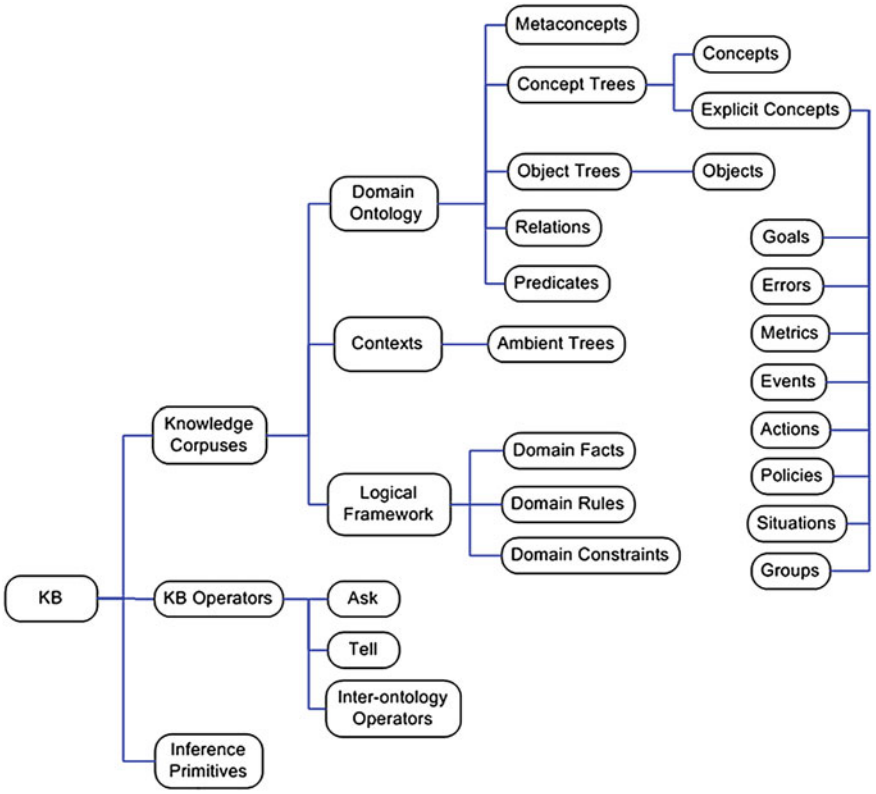

Fig. 2.7 KnowLang specification model

A KnowLang ontology is specified over *concept trees* (data classes, similar to classes in a domain model), *object trees*, *relations*, and *predicates* (Fig. 2.7). Each concept is specified with special properties and functionalities and is hierarchically linked to other concepts through $PARENTS$ and $CHILDREN$ relationships. In addition, for reasoning purposes every concept specified with KnowLang has an intrinsic $STATES$ attribute that may be associated with a set of possible state values the concept instances might be in [49, 50]. The concept instances are considered as objects and are structured in object trees. The latter are a conceptualization of how objects existing in the world of interest are related to each other. The relationships in an *object tree* are based on the principle that objects have properties, where the value of a property is another object, which in turn also has properties.

Figure 2.8 depicts the graphical representation of a concept tree specified with KnowLang. In KnowLang, concepts and objects might be connected via *relations* expressing *relation requirements*. Relations connect two concepts, two objects, or an object with a concept and may have probability-distribution attribute (e.g., over time, over situations, over concepts' properties, etc.). Probability distribution is provided

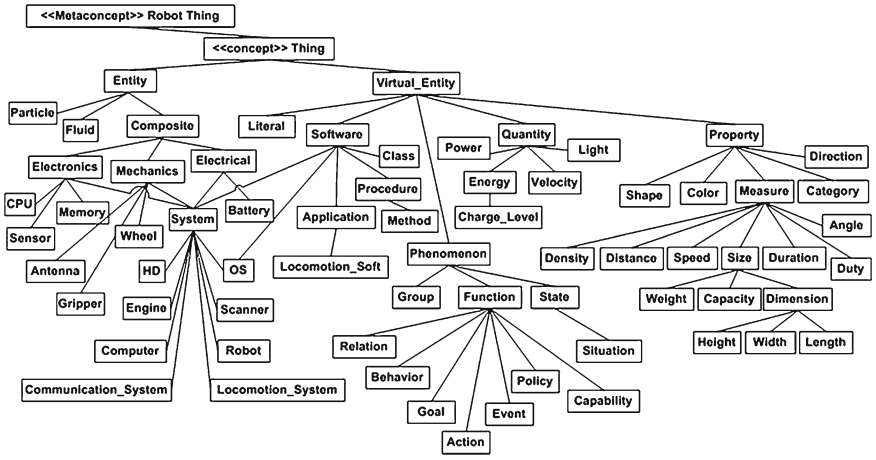

Fig. 2.8 KnowLang ontology specification sample

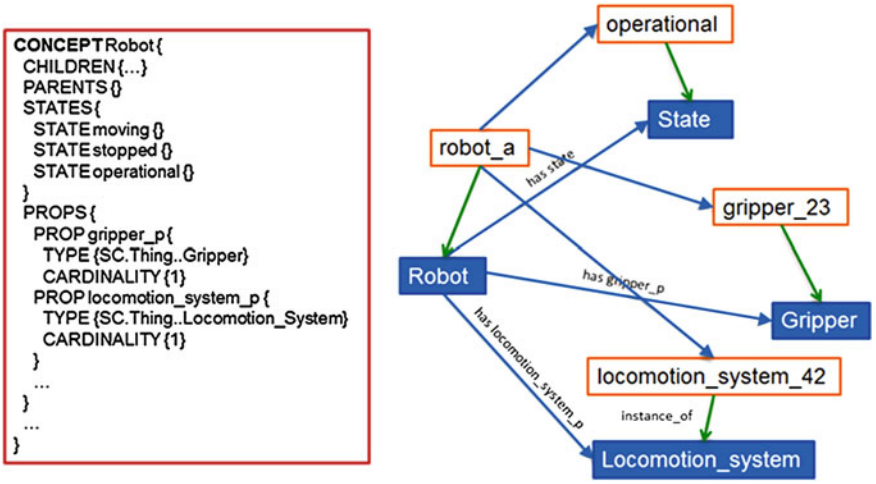

Fig. 2.9 KnowLang specification sample—concept and concept map

to support probabilistic reasoning and by specifying relations with *probability distributions* we actually specify Bayesian networks connecting the concepts and objects of an ontology. Figure 2.9 shows a KnowLang specification sample demonstrating both the language syntax [53] and its visual counterpart—a concept map based on inter-relations with no probability distributions. In general, modeling knowledge with KnowLang goes over a few phases:

1. Initial knowledge requirements gathering—involves domain experts to determine the basic notions, relations and functions (operations) of the domain of interest.

2. Behavior definition—identifies situations and behavior policies as "control data" helping to identify important self-adaptive scenarios.
3. Knowledge structuring—encapsulates domain entities, situations and behavior policies into KnowLang structures like concepts, properties, functionalities, objects, relations, facts and rules.

2.5.4.2 Capturing Autonomy Requirements with KnowLang: Modeling Self-Adaptive Behavior

KnowLang employs special knowledge structures for capturing self-* autonomy requirements (Sects. 2.2.3 and 2.3) by modeling *autonomic self-adaptive behavior* [51]. Such a behavior can be expressed via KnowLang policies, events, actions, situations and relations between policies and situations (Definitions 2.1–2.8). Policies (Π) are at the core of autonomic behavior (autonomic behavior can be associated with autonomy requirements). A policy π has a goal (g), policy situations (Si_π), policy-situation relations (R_π), and policy conditions (N_π) mapped to policy actions (A_π) where the evaluation of N_π may eventually (with some degree of probability) imply the evaluation of actions (denoted with $N_\pi \overset{[Z]}{\rightarrow} A_\pi$ (Definition 2.2). A condition is a Boolean function over ontology (Definition 2.4), e.g., the occurrence of a certain event.

Definition 2.1 $\Pi := \{\pi_1, \pi_2, ..., \pi_n\}, n \geq 0$ (Policies)

Definition 2.2 $\pi :=< g, Si_\pi, [R_\pi], N_\pi, A_\pi, map(N_\pi, A_\pi, [Z]) >$
$A_\pi \subset A, N_\pi \overset{[Z]}{\rightarrow} A_\pi$ (A_π—Policy Actions)
$Si_\pi \subset Si, Si_\pi := \{si_{\pi_1}, si_{\pi_2}, ..., si_{\pi_n}\}, n \geq 0$
$R_\pi \subset R, R_\pi := \{r_{\pi_1}, r_{\pi_2}, ..., r_{\pi_n}\}, n \geq 0$
$\forall r_\pi \in R_\pi \bullet (r_\pi :=< si_\pi, [rn], [Z], \pi >), si_\pi \in Si_\pi$
$Si_\pi \overset{[R_\pi]}{\rightarrow} \pi \rightarrow N_\pi$

Definition 2.3 $N_\pi := \{n_1, n_2, ..., n_k\}, k \geq 0$ (Conditions)

Definition 2.4 $n := be(O)$ (Condition—Boolean Expression)

Definition 2.5 $g := \langle \Rightarrow s' \rangle | \langle s \Rightarrow s' \rangle$ (Goal)

Definition 2.6 $s := be(O)$ (State)

Definition 2.7 $Si := \{si_1, si_2, ..., si_n\}, n \geq 0$ (Situations)

Definition 2.8 $si :=< s, A \overset{\leftarrow}{si}, [E \overset{\leftarrow}{si}], A_{si} >$ (Situation)
$A \overset{\leftarrow}{si} \subset A$ ($A \overset{\leftarrow}{si}$—Executed Actions)
$A_{si} \subset A$ (A_{si}—Possible Actions)
$E \overset{\leftarrow}{si} \subset E$ ($E \overset{\leftarrow}{si}$—Situation Events)

Policy situations (Si_π) are situations that may trigger (or imply) a policy π, in compliance with the policy-situations relations R_π (denoted with $Si_\pi \overset{[R_\pi]}{\rightarrow} \pi$), thus implying the evaluation of the policy conditions N_π (denoted with $\pi \rightarrow N_\pi$) (Definition 2.2). Therefore, the optional policy-situation relations (R_π) justify the relationships between a policy and the associated situations (Definition 2.2). In addition, the self-adaptive behavior requires relations to be specified to connect policies with situations over an optional probability distribution (Z) where a policy might be related to multiple situations and vice versa. Probability distribution is provided to support *probabilistic reasoning* and to help the KnowLang Reasoner choose the most probable situation-policy "pair". Thus, we may specify a few relations connecting a specific situation to different policies to be undertaken when the system is in that particular situation and the probability distribution over these relations (involving the same situation) should help the KnowLang Reasoner decide which policy to choose (denoted with $Si_\pi \overset{[R_\pi]}{\rightarrow} \pi$—Definition 2.2).

A goal g is a desirable transition to a state or from a specific state to another state (denoted with $s \Rightarrow s'$) (Definition 2.5). A state s is a Boolean expression over ontology ($be(O)$) (Definition 2.6), e.g., "a specific property of an object must hold a specific value". A situation is expressed with a state (s), a history of actions ($A \overset{\leftarrow}{si}$) (actions executed to get to state s), actions A_{si} that can be performed from state s and an optional history of events $E \overset{\leftarrow}{si}$ that eventually occurred to get to state s (Definition 2.8).

Ideally, policies are specified to handle specific situations, which may trigger the application of policies. A policy exhibits a behavior via actions generated in the environment or in the system itself. Specific conditions determine, which specific actions (among the actions associated with that policy—Definition 2.2) shall be executed. These conditions are often generic and may differ from the situations triggering the policy. Thus, the behavior not only depends on the specific situations a policy is specified to handle, but also depends on additional conditions. Such conditions might be organized in a way allowing for synchronization of different situations on the same policy. When a policy is applied, it checks what particular conditions are met and performs the mapped actions (see $map(N_\pi, A_\pi, [Z])$)—Definition 2.2). An optional probability distribution can additionally restrict the action execution. Although initially specified, the probability distribution at both mapping and relation levels is recomputed after the execution of any involved action. The re-computation is based on the consequences of the action execution, which allows for reinforcement learning.

2.5.4.3 Probability Assessment

The KnowLang approach to modeling self-adaptive behavior requires computation of probability values for ending in possible states when particular actions are executed. In this subsection, we present a *model for assessing probability* applicable to the computation of such probability values. In our approach, the probability assessment is an indicator of the number of possible execution paths an AS (e.g., ExoMars) may

take, meaning the amount of certainty (excess entropy) in the autonomic behavior. To assess that behavior prior to implementation, it is important to understand the interactions among the system components and also the complex interactions with the surrounding environment (space). This can be achieved by modeling the behavior of the *individual reactive components* and the behavior of the *environment factors* (e.g., solar storm, gravity of a planet, etc.), together with the global system behavior as Discrete Time Markov Chains [18], and by assessing the level of probability through calculating the probabilities of the state transitions in the corresponding models. We assume that the component interactions and the environment-system interaction are stochastic processes where the events are not controlled by the AS and thus, their probabilities are considered equal.

The theoretical foundation for our Probability Assessment Model is the property of Markov chains, which states that, given the current state of the whole spacecraft system, its future evolution is independent of its history, which is also the main characteristic of a reactive and autonomic spacecraft [57, 59].

An algebraic representation of a Markov chain is a matrix (called transition matrix) (Table 2.2) where the rows and columns correspond to the states, and the entry p_{ij} in the ith row, jth column is the transition probability of being in state s_j at the stage following state s_i. We need to build such a transition matrix taking into account both the system components and environment factors influencing the system behavior. The following property holds for the calculated probabilities:

$$\sum_j z_{ij} = 1$$

We contend that probability should be calculated from the steady state of the Markov chain. A *steady state* (or equilibrium state) is one in which the probability of being in a state before and after a transition is the same as time progresses. Here, we define probability for a spacecraft system composed of k components and taking into account x environment factors as the level of certainty quantified by the source excess entropy, as follows:

Table 2.2 Transition matrix Z

	s_1	s_2	...	s_i	...	s_n
s_1	p_{11}	p_{12}	...	p_{1j}	...	p_{1n}
s_2	p_{21}	p_{22}	...	p_{2j}	...	p_{2n}
...	...	...	...	...	...	...
s_i	p_{i1}	p_{i2}	...	p_{ij}	...	p_{in}
...	...	...	...	...	...	...
s_n	p_{n1}	p_{n2}	...	p_{nj}	...	p_{nn}

$$
\begin{aligned}
P_{\mathrm{SCE}} &= \sum\nolimits_{i=1,k} H_i + \sum\nolimits_{e=1,x} H_e - H \\
H_i &= -\sum\nolimits_j p_{ij} \log_2(p_{ij}) \\
H_e &= -\sum\nolimits_j p_{ej} \log_2(p_{ej}) \\
H &= -\left(\sum\nolimits_i v_i \sum\nolimits_j p_{ij} \log_2(p_{ij}) + \sum\nolimits_e v_e \sum\nolimits_j p_{ej} \log_2(p_{ej})\right)
\end{aligned}
$$

Here,

- H is an entropy that quantifies the level of uncertainty in the Markov chain corresponding to the entire AS system;
- H_i is a level of uncertainty in a Markov chain corresponding to an autonomic system component;
- H_e is a level of uncertainty in a Markov chain corresponding to an environmental factor, e.g., distance to ground base, solar storm, gravity force of a planet, etc.;
- v is a steady state distribution vector for the corresponding Markov chain;
- p_{ij} values are transition probabilities in the extended state machines modeling the behavior of the i-th component;
- p_{ej} values are transition probabilities in the extended state machines modeling the behavior of the eth environmental factor.

Note that for a transition matrix P, the steady state distribution vector v satisfies the property $v * P = v$, and the sum of its components v_i is equal to 1.

2.5.4.4 Building Deliberative Controllers with KnowLang

KnowLang provides both the specification structures and runtime mechanism (reasoner) for the development of *deliberative controllers* (Sect. 2.4.2.1). It provides for a comprehensive knowledge representation mechanism and support to both logical and statistical reasoning based on integrated Bayesian networks. Moreover, similar to ASSL, KnowLang also implies layering (i.e., it might be used to handle *layered controller architectures*—Sect. 2.4.2.3) for structuring functionalities and computational structures used for reasoning purposes in goal-oriented autonomy. KnowLang provides a deliberative architecture controller through its KnowLang Reasoner. The reasoner is supplied as a component hosted by the autonomous system and thus, it runs in the system's Operational Context as any other system's component. However, it operates in the Knowledge Representation Context (KR Context) and on the KR symbols (represented knowledge). The system talks to the reasoner via special ASK and $TELL$ Operators allowing for knowledge queries and knowledge updates (Fig. 2.10). Upon demand, the KnowLang Reasoner can also build up and return a self-adaptive behavior model—a chain of actions to be realized in the environment or in the system.

KnowLang provides for a predefined set of ASK and $TELL$ Operators allowing for communication with the KB. $TELL$ Operators feed the KR Context with

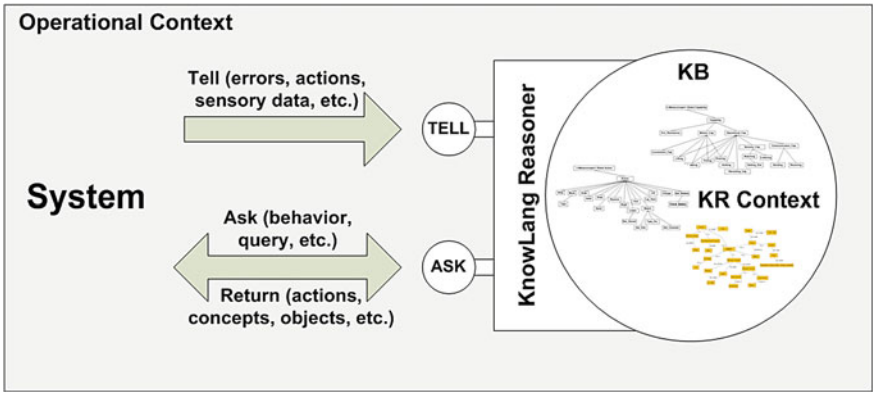

Fig. 2.10 KnowLang reasoner

important information driven by errors, executed actions, new sensory data, etc., thus helping the KnowLang Reasoner update the KR with recent changes in both the system and execution environment. The system uses ASK Operators to receive recommended behavior where knowledge is used against the perception of the world to generate appropriate actions in compliance to some goals and beliefs. In addition, ASK Operators may provide the system with awareness-based conclusions about the current state of the system or the environment and ideally with behavior models for self-adaptation.

2.6 Case Studies: Specifying Autonomy Requirements

In this section, we present three case studies where KnowLang and ASSL have been used to capture autonomy requirements. The first two examples are theoretical models of E4-level autonomic behavior developed with KnowLang and the third one is a concrete example where ASSL has been used to handle E3-level autonomy.

2.6.1 Handling Autonomy Requirements with KnowLang

KnowLang can be used as a *goal-oriented* approach to handling autonomy requirements. By specifying with KnowLang goals and policies handling those goals, actually, we express autonomy requirements (Sect. 2.5.4.2). Note that KnowLang can be successfully used to handle autonomy requirements for all four levels of autonomy recognized by ESA, including the E4 level, which is considered to represent *goal-oriented autonomy* (Sect. 2.1.2).

Fig. 2.11 Elements of the ExoMars program 2016–2018, picture courtesy of ESA

2.6.1.1 Case Study: Autonomy Requirements for ExoMars

To illustrate autonomic self-adaptive behavior based on this approach, we are going to elaborate on the ExoMars (Fig. 2.11) case study by assuming that the robotized rover discovers an interesting rock and it receives a command from the Earth to go and take samples from it.

Eventually, after receiving that command, the rover lost communication with the base on Earth and subsequently starts operating in autonomous mode following requirements for autonomy behavior expressed with KnowLang.

Let us assume that we have used KnowLang to specify autonomy requirements for ExoMars where in addition to another explicit knowledge, we have also specified policy $p1$ (see the KnowLang code below). Although we are missing the basic specification of the involved actions, goal, situation and relation, we can conclude that the current situation $si1$: *a massive rock has been discovered* will trigger a policy $p1$: *go to the rock location* if the relation $r1(si1, p1)$ has the higher probabilistic belief rate.

```
CONCEPT_POLICY p1 { //go to rock
  CHILDREN {} PARENTS {SC.Thing..Policy}
  SPEC {
    POLICY_GOAL {Goal.g1} //get to the rock location
    POLICY_SITUATIONS {Situation.si1} //rock is discovered
    POLICY_RELATIONS {Relation.r1} //relates p1 and si1
    POLICY_ACCTIONS {Action.Turn, Action.Move}
```

```
    POLICY_MAPPINGS {
      MAPPING {
        CONDITIONS {ExoMars.Battery.level >= 0.5 AND Action.GetPriorityTasks (ExoMars) = 0}
        DO_ACTIONS { Action.Turn(Action.GetObjectAngle), Action.Move}
        PROBABILITY {1}
      }
    }
  }
}

CONCEPT_POLICY p2 { //avoid obstacle
  CHILDREN {}
  PARENTS {SC.Thing..Policy}
  SPEC {
    POLICY_GOAL {Goal.g2} //free road
    POLICY_SITUATIONS {Situation.si2} //road is blocked
    POLICY_RELATIONS {Relation.r1} //relates p2 and si2
    POLICY_ACCTIONS {Action.TurnRight, Action.TurnLeft, Action.Move}
    POLICY_MAPPINGS {
      MAPPING {
        DO_ACTIONS {Action.TurnRight, Action.Move}
        PROBABILITY {0.6}
      }
      MAPPING {
        DO_ACTIONS {Action.TurnLeft, Action.Move}
        PROBABILITY {0.4}
      }
    }
  }
}
```

The $p1$ policy will realize actions $Turn$ and $Move$ iff the rover's battery is charged at least 50 % and there is no another higher priority task to finish up first (currently ongoing or scheduled). Ideally, the autonomic behavior will be produced by a sequence of actions, e.g., $\{Action.Turn(Action.GetSignalAngle), Action.Move\}$.

ExoMars will perform the generated actions and will start moving towards the rock. Let us assume that while moving, at certain point, the rover will hit a crack in the terrain and get into a situation $si2$: *road is blocked*, which by specification is related to policy $p2$: *avoid obstacle* (see KnowLang code above). Policy $p2$ will force the rover to turn right and move, because of the initial probability distribution in the $MAPPING$ sections (see KnowLang code above). Eventually, the rover will reach a bridge in the crack and thus, will accomplish the $p2$'s goal $g2$: *free road*. Then it will go back to the initial situation $si1$: *a massive rock has been discovered*, which will trigger the policy $p1$: *go to the rock location* and the robot will start moving again towards the located rock.

Let us suppose that there are more cracks on the route to the located rock and any time when ExoMars gets into situation $si2$: "*road is blocked*" it will continue applying the $p2$ policy by avoiding the crack from the right side until it hits a very long crack on the right side and gets into a situation $si3$: "*tracked object is lost*". This new situation shall trigger another policy $p3$: "*go back until object appears*", which will move the robot back to a point where the rock can be located again and then, the robot will get back to situation $si2$ and policy $p2$. Following $p2$, the robot can fall again into $si3$ and then back to $si2$. However, every time when policy $p2$ fails to accomplish its goal $g2$: "*free road*", the KnowLang Reasoner re-computes the probability distribution in the $MAPPING$ sections (see KnowLang code above), which eventually may lead to a point where by applying policy $p2$ the robot will turn left and move, i.e., it will self-adapt to the current situation and will try to avoid the crack from the left side.

Note that in this case study, we presented adaptation at the level of conditional mapping within a policy as presented in the formal KnowLang model for self-adaptive behavior (Sect. 2.5.4.2).

2.6.1.2 Case Study: Autonomy Requirements for a Transportation Robot

In this case study, we demonstrate how KnowLang can be used to handle autonomy requirements for a transportation robot carrying items from point A to point B by using two possible routes—route one and route two (Fig. 2.12).

Similar to the previous case study (Sect. 2.6.1.1), we specify autonomy requirements to handle autonomous behavior by specifying with KnowLang policies, goals, and situations, together with the accompanying actions, events, etc. Let's assume

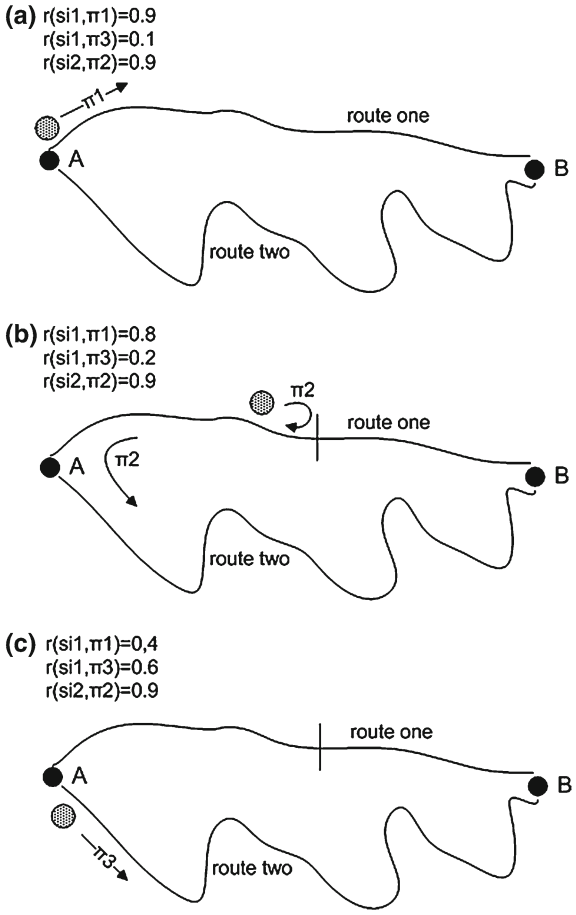

Fig. 2.12 Transportation robot case study

that we specified a situation $si1$: "*robot is in point A and loaded with items*", which will trigger a policy $\pi 1$:"*go to point B via route one*" if the relation $r(si1, \pi 1)$ has the higher probabilistic belief rate (let's assume that such a rate has been initially given to this relation because route one is shorter—Fig. 2.12a).

Any time when the robot gets into situation $si1$ it will continue applying the $\pi 1$ policy until it gets into a situation $si2$: "*route one is blocked*" while applying that policy. The $si2$ situation will trigger a policy $\pi 2$: "*go back to* $si1$ *and then apply policy* $\pi 3$" (Fig. 2.12b). Policy $\pi 3$ is defined as $\pi 3$: "*go to point B via route two*". The unsuccessful application of policy $\pi 1$ will decrease the probabilistic belief rate of relation $r(si1, \pi 1)$ and the eventual successful application of policy $\pi 3$ will increase the probabilistic belief rate of relation $r(si1, \pi 3)$ (Fig. 2.12b). Thus, if route one continues to be blocked in the future, the relation $r(si1, \pi 3)$ will get to have a higher probabilistic belief rate than the relation $r(si1, \pi 1)$ and the robot will change its behavior by choosing route two as a primary route (Fig. 2.12c). Similarly, this situation can change in response to external stimuli, e.g., route two got blocked or a "*route one is obstacle-free*" message is received by the robot.

Note that in this case study, we presented adaptation at the level of situation-policy relations as presented in the formal KnowLang model for self-adaptive behavior (Sect. 2.5.4.2).

2.6.2 Specifying Autonomy Requirements for Voyager with ASSL

In this subsection, we demonstrate how the ASSL framework can be used to specify *self-* requirements* for an unmanned mission. In this exercise, we explore the image-processing system implemented onboard the Voyager spacecraft [41]. In order to take pictures, Voyager II carries two television cameras onboard—one for wide-angle images and one for narrow-angle images, where each camera records images with a resolution of 800×800 pixels. Both cameras can record images in black-and-white only, but each camera is equipped with a set of color filters, which helps in the reconstruction of images be as fully-colored ones. To transmit pictures to Earth, Voyager II uses its 12-foot dish antenna [41] to send streams of pixels. It uses the same microwave frequencies used for radar. However, due to the long distance and to fundamental laws of physics, the strength of the radio signal is diminished proportionally and it reaches antennas on Earth with a strength 20 billion times weaker [8]. To counter this, the signals are received by a network of enormous antennas located in Australia, Japan, California, and Spain. Next, all the faint signals received from Voyager II are combined and processed by the Voyager Mission base on Earth to reduce electronic noise, blend, and filter the composed pictures.

2.6.2.1 Voyager Image-Processing Behavior Algorithm

An autonomous-specific behavior is observed in the Voyager spacecraft when a picture must be taken and sent to Earth. The following elements describe the algorithm we apply to specify the image-processing behavior observed in the Voyager mission with ASSL.

1. The Voyager II spacecraft:
   (a) uses its cameras to monitor space objects and decide when it is time to take a picture;
   (b) takes a picture with its wide-image camera or with its narrow-image camera;
   (c) notifies the antennas on Earth with "image session start" messages that an image transmission is about to start;
   (d) applies each color filter and sends the stream of pixels for each filter to Earth;
   (e) notifies antennas on Earth for the end of each session with "image session end" messages.
2. The antennas on Earth:
   (a) are prompted to receive the image by the "image session start" messages (one per applied filter);
   (b) receive image pixels;
   (c) are prompted to terminate the image sessions by "image session end" messages;
   (d) send the collected images to the Voyager Mission base on Earth.
3. The Voyager Mission base on Earth receives the image messages from the antennas.

2.6.2.2 Specifying Voyager Mission with ASSL

In order to specify the algorithm described in Sect. 2.4.2.1, we apply the ASSL multi-tier specification model (Sect. 2.5.3) and specified the Voyager II Mission at the three main ASSL tiers—AS (autonomic system) tier, $ASIP$ (autonomic system specification protocol) tier, and AE (autonomic element) tier. Hence, in our specification, we specify the Voyager II spacecraft and the antennas on Earth as AEs (autonomic elements) that follow their encoded autonomic behavior and exchange predefined ASSL messages over predefined ASSL communication channels. The Voyager mission's autonomic behavior is specified at both AS and AE tiers as a *self-management policy* called $IMAGE\_PROCESSING$. Thus, the global autonomic behavior of the Voyager II Mission is determined by the specification of that policy at each AE and at the global AS tier. The full specification is presented in Appendix A.

AS Tier Specification

At this tier, we specify the global AS-level autonomic behavior of the Voyager Mission. This behavior is encoded in the specification of an $IMAGE\_PROCESSING$

self-management policy. At this tier, that policy specifies an image-receiving process taking place at the four antennas on Earth (located in Australia, Japan, California, and Spain). In fact, as specified at the AS Tier, this policy forms the autonomic image-processing behavior of the Voyager Mission base on Earth. Here, we specified four "*inProcessingImage_*" fluents (one per antenna), which are initiated by events prompted when an image has been received, and terminated by events prompted when the received image has been processed. Further, all the four fluents are mapped to a *processImage* action. The following specification sample shows a fluent specification together with its mapping:

```
FLUENT inProcessingImage_AntSpain {
 INITIATED_BY { EVENTS.imageAntSpainReceived }
 TERMINATED_BY { EVENTS.imageAntSpainProcessed }
}
MAPPING {
 CONDITIONS { inProcessingImage_AntAustralia }
 DO_ACTIONS { ACTIONS.processImage(''Antenna_Australia'') }
}
```

Here, the specification of the events that initiate and terminate that fluent is the following:

```
EVENT imageAntSpainReceived {
 ACTIVATION { RECEIVED { ASIP.MESSAGES.msgImageAntSpain }
 }
}
EVENT imageAntSpainProcessed { }
```

Note that the *processImage* action is an *IMPL* action [54, 55], i.e., it is a kind of abstract action that does not specify any statements to be performed. The ASSL framework considers the *IMPL* actions as "*to be manually implemented*" after code generation. The following is a partial specification of that action:

```
ACTION IMPL processImage {
 PARAMETERS { string antennaName }
 GUARDS {
  ASSELF_MANAGEMENT.OTHER_POLICIES.IMAGE_PROCESSING.inProcessingImage_AntAustralia
   OR
  ASSELF_MANAGEMENT.OTHER_POLICIES.IMAGE_PROCESSING.inProcessingImage_AntJapan
  ....
 }
 TRIGGERS {
  IF antennaName = ''Antenna_Australia'' THEN
   EVENTS.imageAntAustraliaProcessed
  END
  ELSE ....
 }
}
```

Here, the *processImage* action is specified to accept a single parameter. The latter allows that action to process images from all four antennas. Moreover, there is a special *GUARDS* clause that is specified to prevent execution of the action when none of the four fluents is initiated. The action triggers an *imageAnt*[antenna name]*Processed* event if the action is performed with no exceptions.

ASIP Tier Specification

At this tier, we specify the AS-level communication protocol—the autonomic system interaction protocol (ASIP) (Sect. 2.5.3). This communication protocol is specified to be used by the four antennas when these communicate with the Voyager Mission base on Earth. Here, at this tier we specify four image messages (one per antenna),

a communication channel that is used to communicate these messages, and communication functions (e.g., $sendImageMsg$ and $receiveImageMsg$) to send and receive these messages over that communication channel. Note that the communication functions accept a parameter that allows same communication functions to send or receive messages to and from different antennas. Please refer to Appendix A for the ASSL specification of the Voyager ASIP.

AE Tier Specification

At this tier, we specified five AEs. The Voyager II spacecraft and all four antennas on Earth (the antennas located in Australia, Japan, California, and Spain), are specify as AEs. Note that here, we specify the $IMAGE\_PROCESSING$ self-management policy at the level of single AE and thus, this policy is realized over all AEs specified for the Voyager Mission. The following elements present important details of this specification. Please, refer to Appendix A for the complete specification.

AE Voyager

The most complex AE is the one specified for the Voyager II spacecraft. To express the $IMAGE\_PROCESSING$ self-management policy for this AE, we specified two fluents: $inTakingPicture$ and $inProcessingPicturePixels$. The following ASSL code presents that self-management policy with both fluents and mapping sections.

```
AESELF_MANAGEMENT {
 OTHER_POLICIES {
  POLICY IMAGE_PROCESSING {
   FLUENT inTakingPicture {
    INITIATED_BY { EVENTS.timeToTakePicture }
    TERMINATED_BY { EVENTS.pictureTaken }
   }
   FLUENT inProcessingPicturePixels {
    INITIATED_BY { EVENTS.pictureTaken }
    TERMINATED_BY { EVENTS.pictureProcessed }
   }
   MAPPING {
    CONDITIONS { inTakingPicture }
    DO_ACTIONS { ACTIONS.takePicture }
   }
   MAPPING {
    CONDITIONS { inProcessingPicturePixels }
    DO_ACTIONS { ACTIONS.processPicture }
   }
  }
 }
} // AESELF_MANAGEMENT
```

Here, the $inTakingPicture$ fluent is initiated by a $timeToTakePicture$ event and terminated by a $pictureTaken$ event. This event also initiates the $inProcessingPic\text{-}turePixels$ fluent, which is terminated by the $pictureProcessed$ event. Both fluents are mapped to the actions $takePicture$ and $processPicture$ respectively.

In addition, we specified an $AEIP$ (autonomic element interaction protocol) (Sect. 2.5.3), which is used by the Voyager AE to communicate with the four antenna AEs and to monitor and control the two cameras (wide-image camera and narrow-image camera) on board. Thus, with this AEIP we specified:

- ASSL messages needed to send an image pixel and messages that notify the antenna AEs that an image-receiving session is about to begin or end.
- A private communication channel.

- Three communication functions that send the AEIP messages over the AEIP communication channel.
- Two special managed elements (termed *wideAngleCamera* and *narrowAngleCamera*) to specify interface functions needed by the Voyager AE to monitor and control both cameras. Through their interface functions, both managed elements are used by the actions mapped to the fluents *inTakingPicture* and *inProcessingPicturePixels* to take pictures, apply filters, and detect interesting space objects.

The following specification sample shows a partial specification of one of these managed elements.

```
MANAGED_ELEMENT wideAngleCamera {
 INTERFACE_FUNCTION takePicture { }
 ....
 INTERFACE_FUNCTION countInterestingObjects {
  RETURNS { integer }
 }
}
```

Moreover, an *interestingObjects* metric is specified to count all detected interesting objects, which the Voyager AE takes pictures of. The source of this metric is specified as one of the managed element interface functions (*countInterestingObjects*), i.e., the metric gets updated by that interface function.

```
METRIC interestingObjects {
 METRIC_TYPE { RESOURCE }
 METRIC_SOURCE { AEIP.MANAGED_ELEMENTS.wideAngleCamera.countInterestingObjects }
 THRESHOLD_CLASS { integer [ 0⁻ ) }
}
```

Note that the *timeToTakePicture* event (it activates the *inTakingPicture* fluent) is prompted by a change in this metric's value. Here, in order to simulate this condition, we also activate this event every 60 s on a periodic basis.

```
EVENT timeToTakePicture {
 ACTIVATION { CHANGED { METRICS.interestingObjects } OR PERIOD { 60 SEC } }
}
```

The four antenna AEs are specified as friends (at the *FRIENDS* sub-tier) of the Voyager AE. According to the ASSL semantics [54, 55] friends can share private interaction protocols. Thus, the antenna AEs can use the messages and channels specified by the AEIP of the Voyager AE.

Antenna AEs

We specify the four antennas receiving signals from the Voyager II spacecraft as AEs, i.e., we specified AEs termed *Antenna_Australia*, *Antenna_Japan*, *Antenna_California*, and *Antenna_Spain*. Here, the *IMAGE_PROCESSING* self-management policy for these AEs is specified with a few pairs of *inStartingImageSession* - *inCollectingImagePixels* fluents. A pair of such fluents is specified per image filter and determines states of the antenna AE when an image-receiving session is starting and when the antenna AE is collecting the image pixels.

Because the Voyager AE processes the images by applying different filters and sends each filtered image separately, we specified for each applied filter different fluents in the antenna AEs [52] (see Appendix A for the complete *IMAGE_PROCESSING* specification at the antenna AEs). This allows an antenna AE to process a collection of multiple filtered images simultaneously. Note that according to the ASSL formal semantics, a fluent cannot be re-initiated while it is initiated, thus preventing the same fluent be initiated simultaneously twice or more times [54, 55]. Here, these fluents are initiated and terminated by AE events specified to be prompted by the Voyager AE's messages notifying that an image-receiving session begins or ends. The following partial specification shows two of the *IMAGE_PROCESSING* fluents. These fluents are mapped to AE actions that collect the image pixels per filtered image.

```
FLUENT inStartingGreenImageSession {
 INITIATED_BY { EVENTS.greenImageSessionIsAboutToStart }
 TERMINATED_BY { EVENTS.imageSessionStartedGreen }
}
FLUENT inCollectingImagePixelsBlue {
 INITIATED_BY { EVENTS.imageSessionStartedBlue }
 TERMINATED_BY { EVENTS.imageSessionEndedBlue }
}
```

In addition, an *inSendingImage* fluent is specified. This fluent activates when the antenna AE is done with the image collection work, i.e., all the filtered images (for all the applied filters) have been collected. The fluent is mapped to a sendImage action that sends the filtered images as one image to the Voyager Mission base on Earth. The following listing presents two of the events used to initiate those fluents.

```
EVENT greenImageSessionIsAboutToStart {
 ACTIVATION { SENT { AES.Voyager.AEIP.MESSAGES.msgGreenSessionBeginAus } }
}
EVENT imageSessionStartedBlue {
 ACTIVATION { RECEIVED { AES.Voyager.AEIP.MESSAGES.msgBlueSessionBeginAus } }
}
```

Note that the *greenImageSessionIsAboutToStart* event is prompted when the Voyager's *msgGreenSessionBeginSpn* message has been sent, and the *imageSession-StartedBlue* event is prompted when the Voyager AE's *msgBlueSessionBeginSpn* message has been received by the antenna. Moreover, each antenna AE specifies communication functions which allows the AE to receive the Voyager AE's messages (Appendix A). These communication functions are called by the AE actions.

2.7 Summary

Contemporary software-intensive systems, such as modern spacecraft and unmanned exploration platforms (e.g., ExoMars) generally exhibit a number of autonomic features resulting in complex behavior and complex interactions with the operational environment, often leading to a need of self-adaptation. To properly develop such systems, it is very important to handle the autonomy requirements properly. However, requirements engineering for ASs appears to be a wide open research area with

only a limited number of approaches yet considered. It is our understanding that formal methods can eventually be successful in capturing autonomy requirements. An AS-dedicated formalism might help developers with a well-defined formal semantics that makes the specification of autonomy requirements a base from which developers can design, implement, and verify ASs.

In this chapter, we have presented *generic autonomy requirements* for space missions along with *controller architectures* for robotic systems. Further, we have presented formal methods that cope with both generic autonomy requirements and controller architectures, and as such can lay the foundations of a new AREM (Autonomy Requirements Engineering Model) framework dedicated to autonomic features of software-intensive systems including autonomous space missions. As presented, the targeted AREM approach shall be eventually goal-oriented and based on both KnowLang and ASSL formal methods. As a proof of concept and to demonstrate the expressiveness of the specification languages, we have presented three case studies where KnowLang and ASSL were used to capture to some extend autonomy requirements for both L4 and L3 levels of autonomy.

References

1. Amey, P.: Correctness by construction: better can also be cheaper. CrossTalk Mag. J. Def. Softw. Eng. **2**, 24–28 (2002)
2. Andrei, O., Kirchner, H.: A higher-order graph calculus for autonomic computing. In: Graph Theory, Computational Intelligence and Thought. No. 5420 in Lecture Notes in Computer Science, pp. 15–26. Springer, Heidelberg (2008)
3. Assurance Process for Complex Electronics, NASA: Requirements engineering (2009). http://www.hq.nasa.gov/office/codeq/software/ComplexElectronics/l_requirements2.htm
4. Banâtre, J.P., Fradet, P., Radenac, Y.: Programming self-organizing systems with the higher-order chemical language. Int. J. Unconv. Comput. **3**(3), 161–177 (2007)
5. Beaudette, S.: Satellite Mission Analysis, FDR-SAT-2004-3.2.A. Technical report, Carleton University (2004)
6. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Guernic, P.L., Simone, R.D.: The synchronous languages twelve years later. Proc. IEEE **91**(1), 64–83 (2003)
7. Brachman, R.J., Levesque, H.J.: Knowledge Representation and Reasoning. Elsevier, San Francisco (2004)
8. Browne, W.M.: Technical "Magic" Converts a Puny Signal into Pictures. NY Times (1989)
9. Cheng, B., Atlee, J.: Research directions in requirements engineering. In: Proceedings of the 2007 Conference on Future of Software Engineering (FOSE 2007), pp. 285–303. IEEE Computer Society, Los Alamitos (2007)
10. Cheng, S.W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS'06), pp. 2–8 (2006)
11. Cortim: LEXIOR: LEXIcal analysis for improvement of requirements. www.cortim.com
12. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisitions. Sci. Comput. Program. **20**, 3–50 (1993)
13. Devedzic, V., Radovic, D.: A framework for building intelligent manufacturing systems. IEEE Trans. Syst. Man Cybern. C Appl. Rev. **29**, 422–439 (1999)
14. ECSS Secretariat: Space engineering: space segment operability. Technical report, ESA-ESTEC, Requirements and Standards Division, ECSS-E-ST-70-11C, Noordwijk, The Netherlands (2008)

15. Endsley, M.: Toward a theory of situation awareness in dynamic systems. Hum. Factors **37**(1), 32–64 (1995)
16. ESA: Automatic code and test generation (2007). http://www.esa.int/TEC/Software_engineering_and_standardisation/TECOQAUXBQE_2.html
17. ESA: Requirement engineering and modeling, software engineering and standardization (2007). http://www.esa.int/TEC/Software_engineering_and_standardisation/TECLCAUXBQE_0.html
18. Ewens, W.J., Grant, G.R.: Stochastic processes (i): Poison processes and Markov chains. In: Statistical Methods in Bioinformatics, 2nd edn. Springer, New York (2005)
19. Fickas, S., Feather, M.: Requirements monitoring in dynamic environments. In: Proceedings of the IEEE International Symposium on Requirements Engineering (RE 1995), pp. 140–147. IEEE Computer Society, Los Alamitos (1995)
20. Fortescue, P., Swinerd, G., Stark, J. (eds.): Spacecraft Systems Engineering, 4th edn. Wiley, New York (2011)
21. George, L., Kos, L.: Interplanetary Mission Design Handbook: Earth-to-Mars Mission Opportunities and Mars-to-Earth Return Opportunities 2009–2024. NASA,Marshall Space Flight Center. Springfield, Huntsville (1998)
22. Goldsby, H., Sawyer, P., Bencomo, N., Hughes, D., Cheng, B.: Goal-based modeling of dynamically adaptive system requirements. In: Proceedings of the 15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS). IEEE Computer Society, Los Alamitos (2008)
23. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic Publishers, Boston (1993)
24. IBM Corporation: Policy management for autonomic computing—version 1.2. Technical report, IBM Tivoli (2005)
25. IBM: Autonomic computing: IBM's perspective on the state of information technology, IBM autonomic computing manifesto (2001). http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf
26. IEEE Computer Society: IEEE Standard IEEE-Std-830-1998: IEEE Recommended Practice for Software Requirements Specification (1998)
27. Lapouchnian, A., Yu, Y., Liaskos, S., Mylopoulos, J.: Requirements-driven design of autonomic application software. In: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2006), p. 7. ACM, Boston (2006)
28. Madni, A.: Agiletecting: a principled approach to introducing agility in systems engineering and product development enterprises. J. Integr. Des. Process Sci. **12**(4), 1–7 (2008)
29. Mylopoulos, J., Chung, L., Nixon, B.: Representing and using non-functional requirements: a process-oriented approach. IEEE Trans. Softw. Eng. **18**(6), 483–497 (1992)
30. NASA: Deep Space 1 (2010). http://nmp.jpl.nasa.gov/ds1/
31. NASA, Goddard Space Flight Center: The extended mission Earth Observing-1 (2008). http://eo1.gsfc.nasa.gov/
32. Ocón, J. et al.: Autonomous controller—survey of the state of the art, Version 1.3, GOAC, GMV-GOAC-TN01. Technical report, ESTEC/Contract No. 22361/09/NL/RA (2011)
33. ProForum, W.: Web ProForum tutorials. http://www.iec.org
34. Robinson, W.N.: Integrating multiple specifications using domain goals. In: Proceedings of the 5th International Workshop on Software Specification and Design (IWSSD-5), pp. 219–225. IEEE, New York (1989)
35. Savor, T., Seviora, R.: An approach to automatic detection of software failures in real-time systems. In: Proceedings of the IEEE Real-Time Technology and Applications Symposium, pp. 136–147. IEEE Computer Society, Los Alamitos (1997)
36. Scerri, P., Pynadath, D., Tambe, M.: Towards adjustable autonomy for the real-world. J. Artif. Intell. Res. **17**(1), 171–228 (2002)
37. Schmidt, M., Schilling, K.: Satellite constellations and ground networks—a new perspective on distributed space missions. In: Proceedings of 2nd Nano-Satellite Symposium, Tokyo, Japan (2011)

38. Simon, H.: The Sciences of the Artificial, 2nd edn. MIT Press, Cambridge (1981)
39. Soutchanski, M.: High-level robot programming and program execution. In: Proceedings of the ICAPS'03 Workshop on Plan Execution. AAAI Press, Cambridge (2003)
40. Sutcliffe, A., Fickas, S., Sohlberg, M.: PC-RE a method for personal and context requirements engineering with some experience. Requir. Eng. J. **11**, 1–17 (2006)
41. The Planetary Society: Space topics: Voyager—the story of the mission (2010). http://planetary.org/explore/topics/space_missions/voyager/objectives.html
42. Truszkowski, W., Hallock, L., Rouff, C., Karlin, J., Rash, J., Hinchey, M., Sterritt, R.: Autonomous and Autonomic Systems—with Applications to NASA Intelligent Spacecraft Operations and Exploration Systems. Springer, Berlin (2009)
43. UBM Tech: Cittio's WatchTower 3.0 (2014). http://www.networkcomputing.com/careers-and-certifications/cittios-watchtower-30/d/d-id/1218255?
44. University of Chicago: Nimbus (2014). http://www.nimbusproject.org
45. van Lamsweerde, A., Darimont, R., Letier, E.: Managing conflicts in goal-driven requirements engineering. In: IEEE Transactions on Software Engineering, Special Issue on Inconsistency Management in Software Development (1998)
46. van Lamsweerde, A.: Requirements engineering in the year 00: a research perspective. In: Proceedings of the 22nd IEEE International Conference on Software Engineering (ICSE-2000), pp. 5–19. ACM, Boston (2000)
47. Vassev, E., Hinchey, M., Balasubramaniam, D., Dobson, S.: An ASSL approach to handling uncertainty in self-adaptive systems. In: Proceedings of the 34th annual IEEE Software Engineering Workshop (SEW34), pp. 11–18. IEEE Computer Society, Los Alamitos (2011)
48. Vassev, E., Hinchey, M., Paquet, J.: Towards an ASSL specification model for NASA swarm-based exploration missions. In: Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008)—AC Track, pp. 1652–1657. ACM, Boston (2008)
49. Vassev, E., Hinchey, M.: Knowledge representation and awareness in autonomic service-component ensembles—state of the art. In: Proceedings of the 14th IEEE International Symposium on Object/Component/ Service-Oriented Real-time Distributed Computing Workshops, pp. 110–119. IEEE Computer Society, Los Alamitos (2011)
50. Vassev, E., Hinchey, M.: Knowledge representation for cognitive robotic systems. In: Proceedings of the 15th IEEE International Symposium on Object/Component/Service-Oriented Real-time Distributed Computing Workshops (ISCORCW 2012), pp. 156–163. IEEE Computer Society, Los Alamitos (2012)
51. Vassev, E., Hinchey, M.: Knowledge-based self-adaptation. In: Proceedings of the 6th Latin-American Symposium on Dependable Computing (LADC 2013), Rio de Janeiro, Brazil, pp. 11–18. SBC - Brazilian Computer Society Press, Rio de Janeiro (2013)
52. Vassev, E., Hinchey, M.: Modeling the image-processing behavior of the NASA Voyager Mission with ASSL. In: Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'09), pp. 246–253. IEEE Computer Society, Los Alamitos (2009)
53. Vassev, E.: KnowLang grammar in BNF, lero-tr-2012-04. Technical report, Lero, University of Limerick (2012)
54. Vassev, E.: Towards a framework for specification and code generation of autonomic systems. Ph.D. thesis, Computer Science and Software Engineering Department, Concordia University, Quebec, Canada (2008)
55. Vassev, E. (ed.): ASSL: Autonomic System Specification Language—A Framework for Specification and Code Generation of Autonomic Systems. LAP Lambert Academic Publishing, Saarbrucken (2009)
56. Vassev, E., Hinchey, M.: ASSL: a software engineering approach to autonomic computing. IEEE Comput. **42**(6), 106–109 (2009)
57. Vassev, E., Hinchey, M.: The challenge of developing autonomic systems. IEEE Comput. **43**(12), 93–96 (2010)
58. Vassev, E., Hinchey, M.: Awareness in software-intensive systems. IEEE Comput. **45**(12), 84–87 (2012)

59. Vassev, E., Sterritt, R., Rouff, C., Hinchey, M.: Swarm technology at NASA: building resilient systems. IT Prof. **14**(2), 36–42 (2012)
60. Wei, C., Hindriks, K.V.: An agent-based cognitive robot architecture. In: Proceedings of Programming Multi-Agent Systems (ProMAS) Workshop Affiliated with AAMAS 2012, Valencia, Spain, pp. 55–68 (2012)
61. Wertz, J., Larson, W. (eds.): Space Mission Analysis and Design, 3rd edn. Microcosm Press, Dordrecht (1999)
62. Wood, L.: Satellite constellation networks. In: Yongguang Zhang (ed.) Internetworking and Computing Over Satellite Networks, pp. 13–34. Kluwer Academic Publishers, Dordrecht (2003)

http://www.springer.com/978-3-319-09815-9

Autonomy Requirements Engineering for Space Missions

Vassev, E.; Hinchey, M.

2014, XVII, 250 p. 38 illus., Hardcover
ISBN: 978-3-319-09815-9