Autonomy requirements engineering

Emil Vassev, Mike Hinchey

Autonomy Requirements Engineering

Emil Vassev and Mike Hinchey
Lero–the Irish Software Engineering Research Center, University of Limerick, Ireland
emil.vassev@lero.ie, mike.hinchey@lero.ie

Abstract

Contemporary robotics relies on the most recent advances in automation and robotic technologies to promote autonomy and autonomic computing principles to robotized systems. However, it appears that the design and implementation of autonomous systems is an extremely challenging task. The problem is stemming from the very nature of such systems where features like environment monitoring and self-monitoring allow for awareness capabilities driving the system behavior. Moreover, changes in the operational environment may trigger self-adaptation. The first and one of the biggest challenges in the design and implementation of such systems is how to handle requirements specifically related to the autonomy of a system. Requirements engineering for autonomous systems appears to be a wide open research area with only a limited number of approaches yet considered. In this paper, we present an approach to Autonomy Requirements Engineering where goals models are merged with special generic autonomy requirements. The approach helps us identify and record the autonomy requirements of a system in the form of special self- objectives and other assistive requirements, those capturing alternative objectives the system may pursue in the presence of factors threatening the achievement of the initial system goals. The paper presents a case study where autonomy requirements engineering is applied to the domain of space missions.*

Keywords: autonomy requirements, autonomic systems, robotics, requirements engineering, space missions.

1. Introduction

The first step towards development of any software-intensive system is to determine the system's requirements, which includes both requirements elicitation and specification (or modeling). Traditionally, *requirements engineering* is concerned with what a system should do and within which constraints it must do it. To answer these questions, software 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.

Along with the traditional requirements, *requirements engineering for autonomous and self-adaptive systems* (e.g., exploration robots) needs to address requirements related to adaptation issues, in particular: 1) what adaptations are possible; 2) under what constrains; and 3) how those adaptations are realized [1]. Note that adaptations arise when a system needs to cope with changes to ensure realization of the system's objectives.

To handle these and other issues, Lero – the Irish Software Engineering Research Center, is currently conducting a joint project with ESA targeting an *Autonomy Requirements Engineering* (ARE) approach. ARE converts adaptation issues into autonomy requirements where Goal-Oriented Requirements Engineering (GORE) [2] is used along with a model for Generic Autonomy Requirements (GAR) [1]. In the course of this project, ARE was applied to a proof-of-concept case study, to capture autonomy requirements of the ESA's BepiColombo Mission [3]. This paper presents the ARE approach along with the case study where the emphasis is put on the *requirements specification*. The paper is a follow-up to [4] and [5] where we presented our GAR [4] and the ARE process [5] for space missions.

The rest of this paper is organized as follows. Section 2 presents the ARE approach. Section 3 presents the case study where ARE is applied to capture the autonomy requirements of the ESA's BepiColombo Mission with the emphasis put on the requirements specification. Finally, Section 4 presents a brief conclusion and future work.

2. Autonomy Requirements Engineering

ARE should be considered as a *software engineering process* helping to 1) determine what *autonomic features* are to be developed for a particular autonomous system; and 2) generate autonomy requirements supporting those features. A comprehensive and efficient ARE approach should take into account all the autonomy aspects of the targeted system and emphasize the so-called *self-**

requirements by taking into consideration the traditional *functional* and *non-functional* requirements (e.g., safety requirements) [1].

In our approach, ARE: 1) relies on GORE [2] to elicit and define the system goals; and then 2) uses a special framework called Generic Autonomy Requirements (GAR) [1, 4] to derive and define *assistive* and eventually *alternative goals* (or objectives) the system may pursue in the presence of factors threatening the achievement of the initial system goals. In addition, GAR also helps to identify goal-supporting autonomy requirements. Once identified, the autonomy requirements including the self-* objectives might be further specified with formal languages complying with both GORE and GAR (e.g., KnowLang [6]). The outcome of ARE (goals models, requirements specification, etc.) is a precursor of design of autonomic features.

Note that GAR is very generic and needs to be put in the specific system's context and generate generic autonomy requirements for this specific context first, and then merged with the output generated by GORE. For example, as part of this exercise, we put GAR in the context of space missions [4].

2.1. GAR – Generic Autonomy Requirements

GAR considers that the development of autonomous systems is driven by *self-adaptive objectives* and *adaptation-assistive attributes*, which introduce special requirements termed *self-* requirements* [1, 4]:

- **Autonomicity (self-* objectives)** - Autonomicity is one of the essential characteristics of autonomous systems. The self-* objectives provide for autonomous behavior (e.g., self-configuring, self-healing, self-optimizing, and self-protecting).
- **Knowledge** – An autonomous system is intended to possess awareness capabilities based on well-structured knowledge and algorithms operating over the same.
- **Awareness** – A product of knowledge representation, reasoning and monitoring.
- **Monitoring** – The process of obtaining raw data through a collection of sensors or events.
- **Adaptability** – The ability to achieve change in observable behavior and/or structure. Adaptability may require changes in functionality, algorithms, system parameters, or structure. The property is amplified by self-adaptation.
- **Dynamicity** – The technical ability to perform a change at runtime. For example, a technical ability to remove, add or exchange services and components.
- **Robustness** – The ability to cope with errors during execution.
- **Resilience** - A quality attribute prerequisite for resilience and system agility. Closely related to safety, resilience enables systems to bounce back from unanticipated disruptions.
- **Mobility** – A property demonstrating what moves in the system at design time and runtime.

Note that, ARE requires GAR to be put in the operational context of the system in question first (e.g., the BepiColombo space mission), to derive *context-specific GAR* used by the ARE process.

2.2. GORE for ARE

GORE (Goal-Oriented Requirements Engineering) has extended 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, GORE [2] helps engineers *analyze the space of alternatives*, which makes the process of generating functional and non-functional (quality) requirements more systematic in the sense that the designer is exploring an *explicitly represented* space of alternatives.

To comply with ARE, GORE is used to produce *goals models* that represent system objectives and their inter-relationships. Goals are generally modeled with *intrinsic features* such as their type, actors and targets, and with *links* to other goals and to other elements of the requirements model (e.g., constraints). 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 to achieving the high-level goals. ARE merges GORE with a context-specific GAR to arrive at goals models where system goals are supported by self-* objectives promoting autonomicity in system behavior.

3. The BepiColombo Case Study

In this section, we present the BepiColombo case study where ARE is applied to capture the autonomy requirements. The section briefly presents the *requirements elicitation* process (GORE + GAR) and presents in more detail the *requirements specification*. For more details on the requirements elicitation process, please refer to [4] and [5].

3.1. BepiColombo Mission

BepiColombo is an ESA mission to Mercury [3, 7, 8, 9] (see Figure 1) scheduled for launching in 2015.

BepiColombo will perform a series of scientific experiments, tests and measures. For example, BepiColombo will make a complete map of Mercury at different wavelengths. Such a map, will chart the planet's mineralogy and elemental composition. Other experiments will be to determine whether the interior of the planet is molten or not and to investigate the extent and origin of Mercury's magnetic field.

Figure 1. BepiColombo Arriving at Mercury [7]

The space segment of the BepiColombo Mission consists of two orbiters: a *Mercury Planetary Orbiter (MPO)* and a *Mercury Magnetospheric Orbiter (MMO)*. Initially, these two orbiters will be packed together into a special *composite module* used to bring both orbiters into their proper orbits. Moreover, in order to transfer the orbiters to Mercury, the composite module is equipped with an extra electric propulsion module both forming a *transfer module*. The transfer module is intended to do the long cruise from Erath to Mercury by using the electric propulsion engine and the gravity assists of Moon, Venus and Mercury. The transfer module spacecraft will have a 6 year interplanetary cruise to Mercury using solar-electric propulsion and Moon, Venus, and Mercury gravity assists. On arrival in January 2022, the MPO and MMO will be captured into polar orbits. When approaching Mercury in 2022, the *transfer module* will be separated and the *composite module* will use rocket engines and a technique called *weak stability boundary capture* to bring itself into polar orbit around the planet. When the MMO orbit is reached, the MPO will separate and lower its altitude to its own operational orbit. Note that the environment around Mercury imposes strong requirements on the spacecraft design, particularly to the parts exposed to Sun and Mercury: solar array mechanisms, antennas, multi-layer insulation, thermal coatings and radiators.

The Mercury Planetary Orbiter (MPO) is a three-axis-stabilized spacecraft pointing at nadir. The spacecraft shall revolve around Mercury at a relatively low altitude and will perform a series of experiments related to planet-wide remote sensing and radio science. MPO will be equipped with two rocket engines nested in two propulsion modules respectively: a *solar electric propulsion module* (SEPM) and a *chemical propulsion module* (CPM). Moreover, to perform scientific experiments, the spacecraft will carry a highly sophisticated suit of eleven instruments [10].

The Mercury Magnetospheric Orbiter (MMO) is a spin-stabilized spacecraft in a relatively eccentric orbit carrying instruments to perform scientific experiments mostly with fields (e.g., Mercury magnetic field), waves and particles. Similar to MPO, MMO is also equipped with two propulsion modules: a *solar electric propulsion module* (SEPM) and a *chemical propulsion module* (CPM). MMO has altitude control functions, but no orbit control functions. MMO's main structure consists of: *two decks* (upper and lower), a *central cylinder* (thrust tube) and four *bulkheads*. The instruments are located on both decks. The MMO spacecraft will carry five advanced scientific experiments [10].

3.2. ARE – Requirements Elicitation

3.2.1. GORE for BepiColombo. By applying GORE, we build *goals models* that can help us derive and organize the *autonomy requirements* for BepiColombo. In our approach, the models provide the starting point for ARE for BepiColombo by defining 1) *the objectives of the mission* that must be realized in 2) *the system's operational environment* (space, Mercury, proximity to the Sun, etc.), and by identifying the 3) *problems that exist in this environment* as well as 4) *the immediate targets supporting the mission objectives* and 5) *constraints* the system needs to address. Moreover, GORE helps us identify the *mission actors* (mission spacecraft, spacecraft components, environmental entities, base station, etc.). In this exercise, we do not categorize the objectives' actors, but for more comprehensive requirements engineering, actors might be categorized by role or by importance (e.g., main, supporting and offstage actors). Further, the requirements goals models can be used as a *baseline for validating the system*.

Complete goals models along with the accompanying rational can be found in [5]. Due to space limitations, in this paper we present only the *Orbit-placement Goal* [5]:

- **Orbit-placement**: Both MPO and MMO must be placed in orbit around Mercury to fulfill the mission objectives.
  Rationale: When approaching Mercury in, the carrier spacecraft will be separated and the composite spacecraft will use rocket engines and a technique called *weak stability boundary capture* to bring it into polar orbit around the planet. When the MMO orbit is reached, the MPO will separate and lower its altitude to its own operational orbit. Observations from orbit will be taken for at least one Earth year.

Actors: *BepiColombo transfer module, electric propulsion rocket engines, chemical rocket engines, Mercury, the Sun, Base on Earth, BepiColombo composite module (MPO and MMO), MPO, MMO.*
Targets: *MPO orbit, MMO orbit*

3.2.2. GAR for BepiColombo. The BepiColombo Mission falls in the category of *Interplanetary Missions* [4] and consecutively inherits the context-specific GAR model for such missions [4]. A good practice will be to associate the autonomy requirements with each objective (or group of objectives). Thus, we may have autonomy requirements (including self-* objectives) associated with the *Transfer Objective*, the *Orbit-placement Objective* and with the group of *Scientific Objectives* [5]. Due to space limitations, in this paper we present only the autonomy requirements associated with the *Orbit-placement Objective*. For the complete GAR model, refer to [5].

The Orbit-placement Objective is to place both MMO and MPO into their operational orbits around Mercury. When approaching Mercury, the BepiColombo Transfer Module will be separated by releasing the module's SEPM. Then, the BepiColombo Composite Module will use the MMO's rocket engines (mainly the CPM) and the weak stability boundary capture mechanism to move the spacecraft into polar orbit around Mercury (see Section 3). When the MMO orbit is reached, the MPO will separate and lower its altitude to its own operational orbit.

To derive the autonomy requirements assisting that objective, we need to identify the appropriate category of GAR (Generic Autonomy Requirements) that might be applied. Considering the Orbit-placement Objective, the BepiColombo mission falls in the category of *Interplanetary Missions using Low-thrust Trajectories* [4]. Such missions use spacecraft for orbit control activities in geostationary orbits, drag compensation in low orbits, planetary 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* (similar to BepiColombo). Therefore, by considering the Orbit-placement Objective specifics, we derive the autonomy requirements for that objective, by applying GAR for Interplanetary Missions using Low-thrust Trajectories [4]:

- **self-* requirements (autonomicity)**:
  - *self-jettison*: the Transfer Module shall automatically release its SEPM when the right jettison attitude is reached; the Composite Module shall automatically release MMO when the polar orbit is reached.
  - *self-capture*: the Composite Module shall autonomously determine a steering law and use low thrust to achieve capture around Mercury.
  - *self-escape*: the Composite Module shall autonomously acquire the escape procedure and use it to leave Mercury if necessary;
  - *self-low-thrust-trajectory*: autonomously determine a steering law for a thrust vector and use low thrust to bring the Composite Module into polar orbit; autonomously determine a steering law for a thrust vector and use low thrust to bring MPO into its orbit.
  - *self-protection*: both the Composite Module and MPO shall autonomously detect the presence of high solar irradiation and: 1) protect the electronics on board and instruments; 2) get away if possible by using electric propulsion and/or chemical propulsion.
  - *self-thermal-control*: both MMO and MPO shall maintain the onboard equipment and the spacecraft structure in proper temperature range.
  - *self-scheduling*: both the Composite Module and MPO shall autonomously determine what task to perform next in the course of pursuing the Orbit-placement Objective: 1) jettison; 2) start and stop engines; 3) spin-up by using thrusters; 4) moving by using thrusters.
- **knowledge:** *central force field physics; steering law model for weak stability boundary capture; MMO orbit; MPO orbit; maximum rate of change of orbital energy for MMO and MPO; maximum rate of change of orbital inclination for MMO and MPO; instruments onboard together with their characteristics* (acceptable levels of radiation)*; Base on Earth*; *propulsion system* (chemical propulsion rockets)*; communication links*, *data transmission format*, *communication mechanisms onboard*; *gravitational forces* (Sun gravity and Mercury gravity);
- **awareness** (for both the Composite Module and MPO)**:** *Mercury capture awareness; Mercury escape awareness; trajectory velocity awareness; Mercury's magnetic field awareness; Mercury's gravitational force awareness; Sun's gravitational force awareness; awareness of the spacecraft's position on the projected trajectory perturbations; radiation awareness; instrument awareness; sensitive to thermal stimuli; data-transfer awareness; speed awareness; communication awareness*.
- **monitoring** (for both the Composite Module and MPO)**:** *the environment around Mercury* (e.g., radiation level, Mercury, the Sun); *planned operations* (status, progress, feasibility, etc.).
- **adaptability** (for both the Composite Module and MPO)**:** *adapt the low thrust trajectories to orbit and/or altitude perturbations*.

- **dynamicity** (for both the Composite Module and MPO)**:** *dynamic near-body environment*; *dynamic trajectory following procedure*; *dynamic communication links*.
- **robustness** (for both the Composite Module and MPO)**:** *robust to solar irradiation*; *robust to temperature changes* (high temperature amplitude)*; robust to orbit-placement trajectory perturbations; robust to communication losses.*
- **resilience** (for both the Composite Module and MPO)**:** *resilient to magnetic field changes*.
- **mobility** (for both the Composite Module and MPO)**:** *trajectory maneuvers for avoiding orbit and/or altitude perturbations*.

Following the ARE process, next we merge the self-* requirements derived by GAR with the goals models produced by GORE to derive self-* objectives providing *mission behavior alternatives* with respect to the BepiColombo Mission Objectives. The self-* objectives assisting the BepiColombo's Orbit-placement Objective are [5]: *self-jettison* (2 variants), *self-capture*, *self-escape*, *self-low-thrust-trajectory* (2 variants), *self-protection* (4 variants), *self-thermal-control* (2 variants) and *self-scheduling* (3 variants). The self-protection objectives are as following [5]:

- **Self-protection_1**: Autonomously detect the presence of high solar irradiation and protect (eventually turn off or shade) the electronics and instruments on board.
  Actors: *BepiColombo composite module, the Sun, Base on Earth, radiation, shades, power system.*
  Targets: *electronics and instruments*.
- **Self-protection_2**: Autonomously detect the presence of high solar irradiation and get away if possible by using chemical propulsion.
  Actors: *BepiColombo composite module, CPM, Mercury, the Sun, Base on Earth, solar irradiation*.
  Targets: *safe position around Mercury*.
- **Self-protection_3**: Autonomously detect the presence of high solar irradiation and protect (eventually turn off or shade) the electronics and instruments on board.
  Actors: *MPO, the Sun, Base on Earth, solar irradiation, shades, power system.*
  Targets: *electronics and instruments*.
- **Self-protection_4**: Autonomously detect the presence of high solar irradiation and get away if possible by using chemical propulsion.
  Actors: *MPO, CPM, Mercury, the Sun, Base on Earth, solar irradiation*.
  Targets: *safe position around Mercury*.

3.3. ARE – Requirements Specification

The next step after deriving the autonomy requirements per system's objectives (see Section 3.2) shall be their specification, which can be considered as a form of *formal specification* or *requirements recording*. The formal notation to be used for requirements recording must cope with ARE, i.e., it should be expressive enough to handle both the goals models produced by GORE and the requirements generated by GAR. KnowLang [6] is formal method having all the necessary features required to handle such a task. The process of requirements specification 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.

When specifying autonomy requirements with KnowLang, an important factor to take into consideration is to know how the KnowLang framework handles these requirements at runtime. KnowLang comes with a special KnowLang Reasoner [6] that operates on the specified requirements and provides the system with *awareness capabilities*. The reasoner supports both logical and statistical reasoning based on integrated *Bayesian networks*. The KnowLang Reasoner is supplied as a component hosted by the system (e.g., the BepiColombo's MMO spacecraft) 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. Upon demand, the KnowLang Reasoner can also build up and return a *self-adaptive behavior model* as a chain of actions to be realized in the environment or in the system itself [6].

In this section, we present the KnowLang [6] specification of the BepiColombo autonomy requirements. Note that both the specification models and accompanying rationale presented in this section are partial and intended to demonstrate how KnowLang copes with the different autonomy requirements. Moreover, a full specification model of the BepiColombo is too large to be presented here and it is beyond this paper's objectives.

3.3.1. Knowledge. KnowLang [6] is exclusively dedicated to knowledge specification where the latter is specified as a Knowledge Base (KB) comprising a variety of knowledge structures, e.g., *ontologies*, *facts*, *rules*, and

constraints. Here, in order to specify the *autonomy requirements of BepiColombo*, the first step is to specify the KB representing both the *external* (space, Mercury, the Sun, etc.) and *internal* (spacecraft systems - MMO, MPO, etc.) worlds of the BepiColombo Mission. The BepiColombo KB shall contain a few ontologies structuring the knowledge domains of MMO, MPO, BepiColombo Composite Module, BepiColombo Transfer Module, and BepiColombo's operational environment (space) (see Section 3.1). Note that these domains are described via domain-relevant *concepts* and *objects* (concept instances) related through *relations*. To handle explicit concepts like *situations*, *goals*, and *policies*, we grant some of the domain concepts with explicit state expressions (a state expression is a Boolean expression over ontology). Note that being part of the autonomy requirements, knowledge plays a very important role in the expression of the other autonomy requirements: *autonomicity*, *knowledge*, *awareness*, *monitoring*, *adaptability*, *dynamicity*, *robustness*, *resilience*, and *mobility* outlined by GAR (see Section 2.1 and Section 3.2.2).

To express the autonomy requirements of BepiColombo, we specified the necessary knowledge as following. Figure 2, depicts a graphical representation of the *MMO Thing concept tree* relating most of the concepts within the MMO Ontology. Note that the relationships within a concept tree are "is-a" (inheritance), e.g., the *Part* concept is an *Entity* and the *Tank* concept is a *Part* and consecutively *Entity*, etc.

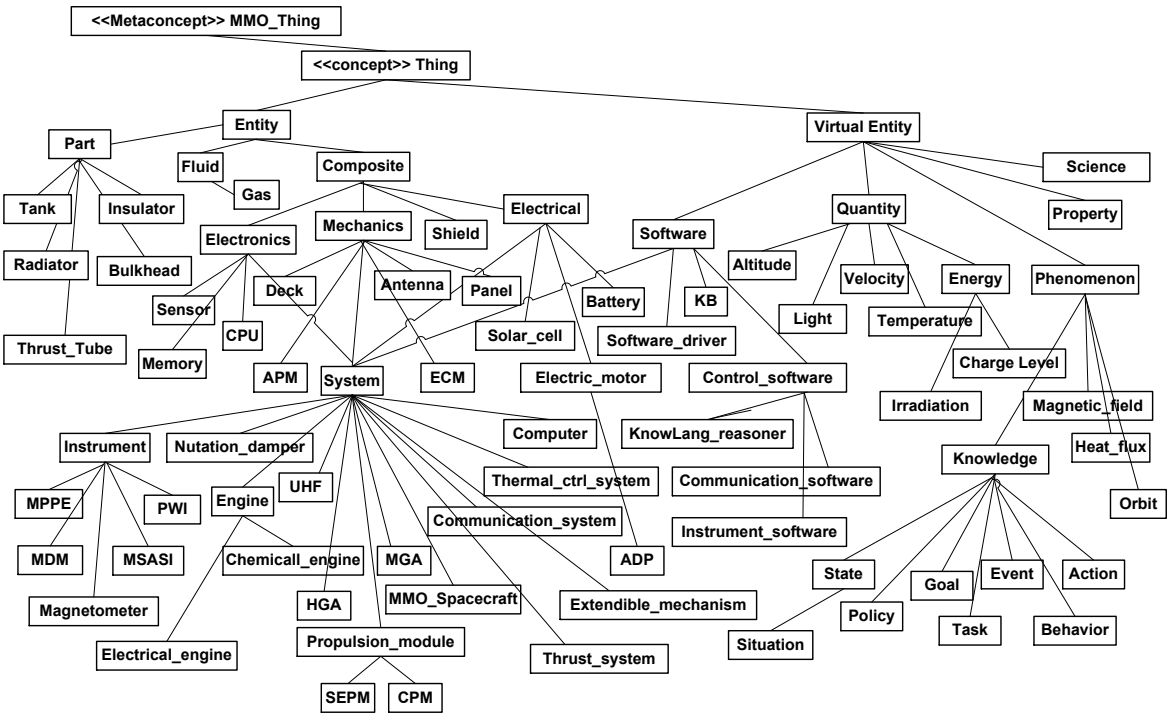

Figure 2. MMO Ontology: MMO_Thing Concept Tree

The following is a sample of the KnowLang specification representing the concepts of the MMO's propulsion modules: *SEPM* and *CPM*. As specified, the concepts in a concept tree might have *properties* of other concepts, *functionalities* (actions associated with that concept), states (Boolean expressions validating a specific state), etc. The *IMPL{}* specification directive references to the implementation of the concept in question, i.e., in the following example *SEPMSystem* is the software implementation (presuming a C++ class) of the MMO's SEPM.

```
CONCEPT SEPM {
   CHILDREN {}
        PARENTS { MMO..System }
        STATES {
            STATE Operational {
            this.solar_cells.Functional AND this.gas_tank.Functional AND
            this.el_engine.Operational AND this.control_soft.Functional }
            STATE Forwarding { IS_PERFORMING(this.forward) }
            STATE Reversing { IS_PERFORMING(this.forward) }
            STATE Started { LAST_PERFORMED(this, this.start) }
            STATE Stopped { LAST_PERFORMED(this, this.stop) }
        }
        PROPS {
    PROP solar_cells {TYPE {MMO..Solar_cell} CARDINALITY {200}}
    PROP gas_tank { TYPE {MMO..Tank} CARDINALITY {1}}
    PROP el_engine { TYPE{MMO..Electrical_Engine} CARDINALITY {1}}
    PROP control_soft {TYPE{MMO..Control_Softare} CARDINALITY {1}}
        }
        FUNCS {
            FUNC reverse { TYPE {MMO..Action.ReverseSEPM } }
            FUNC forward { TYPE {MMO..Action.ForwardSEPM } }
            FUNC start { TYPE {MMO..Action.StartSEPM } }
            FUNC stop { TYPE {MMO..Action.StopSEPM } }
        }
        IMPL { MMO.SEPMSystem } }
```

```
CONCEPT CPM {
    CHILDREN {}
    PARENTS { MMO..System }
    STATES {
      STATE Operational {
      this.gas_tank.Functional AND this.chem_engine.Operational AND
      this.control_soft.Functional }
      STATE Forwarding { IS_PERFORMING(this.forward) }
      STATE Reversing { IS_PERFORMING(this.forward) }
      STATE Started { LAST_PERFORMED(this, this.stop) }
      STATE Stopped { LAST_PERFORMED(this, this.start) }
    }
    PROPS {
      PROP gas_tank { TYPE {MMO..Tank} CARDINALITY {1} }
      PROP chem_engine{TYPE{MMO.Chemcl_Engine} CARDINALITY {1}}
      PROP control_soft{TYPE{MMO.Control_Software} CARDINALITY{1}}
    }
    FUNCS {
      FUNC reverse { TYPE {MMO..Action.ReverseCPM } }
      FUNC forward { TYPE {MMO..Action.ForwardCPM } }
      FUNC start { TYPE {MMO..Action.StartCPM } }
      FUNC stop { TYPE {MMO..Action.StopCPM } }
    }
    IMPL { MMO.CPMSystem }
}
```

As mentioned above, the states are specified as Boolean expressions. For example, the state *Forwarding* is true while the propulsion model is performing the *reverse* function. The KnowLang operator *IS_PERFORMING* evaluates actions and returns true if an action is currently performing. Similarly, the operator *LAST_PERFORMED* evaluates actions and returns true if an action is the last successfully performed action by the concept realization (a concept realization is an object instantiated from that concept, e.g., the *SEPM object* or the *CPM object*). A complex state, might be expressed as a function of other states. For example, the *Operational* state is expressed as a Boolean function of a few other states, particularly, states of the concept properties, e.g., the CPM is operational if its gas *tank* is functional, its *chemical engine* is operational and its *control software* is functional:

```
this.gas_tank.Functional AND this.chem_engine.Operational AND
this.control_soft.Functional
```

As mentioned before, states are extremely important to the specification of goals (objectives), situations, and policies. For example, states help the KnowLang Reasoner determine at runtime whether the system is in a particular situation or a particular goal (objective) has been achieved.

The *MMO_Thing* concept tree (see Figure 2) is the main concept tree of the MMO Ontology. Note that due to space limitations, Figure 2 does not show all the concept tree branches. Moreover, some of the concepts in this tree are "roots" of other trees. For example, the *Action* concept, expressing the common concept for all the actions that can be realized by MMO, is the root of another concept tree (not shown here) where actions are grouped by subsystem. The following is a partial specification of the MMO Spacecraft concept. Note this concept "is-a" system, i.e., it inherits the *System* concept. A system, according to the MMO ontology (see Figure 2) is a complex concept that joins the properties of four other concepts: *Electronics*, *Mechanics*, *Electrical*, and *Software*. Note that to specify MMO states, we used *metrics*. Metrics are intended to handle the *monitoring autonomy requirements* (see Section 3.3.3).

```
CONCEPT MMO_Spacecraft {
      CHILDREN {}
      PARENTS { MMO..System }
      STATES {
          STATE Orbiting {}
          STATE InTransfer {}
          STATE InOrbitPlacement {}
          STATE InJettison {}
STATE InHighIrradiation { MMO..Metric.OutsideRadiation.VALUE > 50 }
STATE InHeatFlux { MMO..Metric.OutsideTemp.VALUE > 150 }
STATE AtPolarOrbit { LAST_PERFORMED(this, this.moveToPolarOrbit) }
STATE ArrivedAtMercury { MMO..Metric.MercuryAltitude.VALUE = 0.39 }
STATE EarthCommunicationLost { MMO..Metric.EarthSignal.VALUE = 0 }  }
      PROPS {
          PROP sepm { TYPE {MMO..SEPM} CARDINALITY {1} }
          PROP cpm { TYPE {MMO..CPM} CARDINALITY {1} }
          PROP upper_deck { TYPE {MMO..Deck} CARDINALITY {1} }
          PROP lower_deck { TYPE {MMO..Deck} CARDINALITY {1} }
          PROP thrust_tube {TYPE {MMO..Thrust_Tube} CARDINALITY {1}}
          PROP bulkhead { TYPE {MMO..Bulkhead} CARDINALITY {4} }
….
      }
      FUNCS {
FUNC moveToPolarOrbit { TYPE {MMO..Action.GoToPolarOrbit} }
FUNC waitForInstrFromEarth { TYPE {MMO..Action.WaitForInstructions} }
      }
      IMPL { MMO.MMOSystem }
}
```

In the KnowLang specification models, we use *concept instances* to represent the real domain entities, e.g., the *MMO antenna*:

```
FINAL OBJECT antenna_1 {
      INSTANCE_OF { MMO..Antenna }  }
```

Note that the *concept instances* are considered as objects, and are structured in *object trees* [6]. 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. Therefore, the *MMO object trees* (due to space limitations, not shown here) are the realization of concepts in the MMO ontology domain. To better understand the relationship between concepts and objects, we may think of concepts as similar to the OOP classes and objects as instances of these classes.

3.3.2. Autonomicity. To specify the *self-*objectives* (autonomicity requirements), we use *goals*, *policies*, and *situations*. These are defined as explicit concepts in KnowLang and for the MMO Ontology we specified them under the concepts *Virtual_entity→Phenomenon→ Knowledge* (see Figure 2). Figure 3, depicts a concept tree

with some of the goals (objectives) related to MMO. Note that most of these goals were directly interpolated from the goals models (see Section 3.2.1) and more specifically, from the *goals model for self-* objectives assisting the Orbit-placement Objective* (see Section 3.2.2).

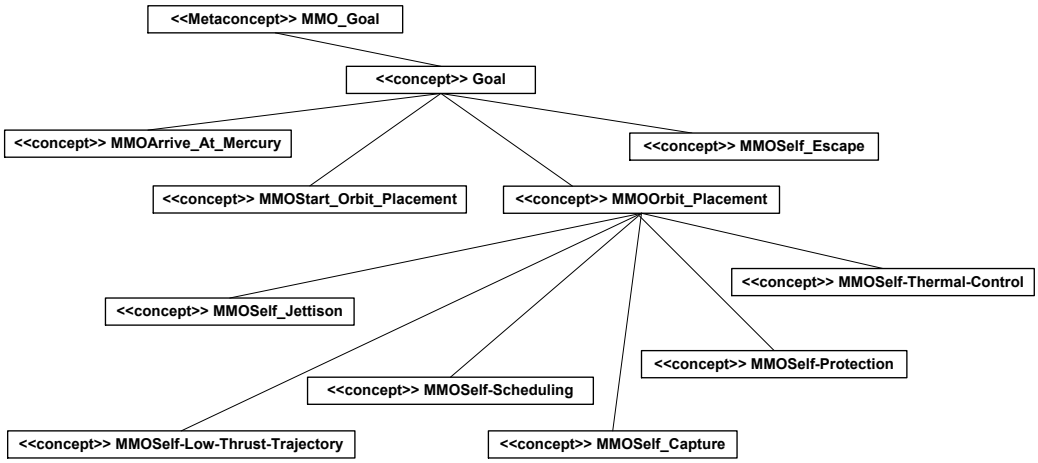

Figure 3. MMO Ontology: MMO_Goal Concept Tree

KnowLang specifies goals as *functions of states* where any combination of states can be involved [6]. A goal has an *arriving state* (Boolean function of states) and an optional *departing state* (another Boolean function of states). A goal with departing state is more restrictive, i.e., it can be achieved only if the system departs from the specific goal's departing state.

The following code samples present the specification of three simple goals. Note that their arriving and departing states are single MMO states, but also can be Boolean functions involving more than one state. Recall that the states used to specify these goals are specified as part of the *MMO_Spacecraft* concept (see Section 3.3.1).

```
CONCEPT_GOAL MMOOrbit_Placement {
    SPEC {
       DEPART { MMO_Spacecraft.STATES.InOrbitPlacement }
       ARRIVE { MMO_Spacecraft.STATES.AtPolarOrbit }
}}
CONCEPT_GOAL MMOArrive_At_Mercury {
    SPEC { ARRIVE { MMO_Spacecraft.STATES.ArrivedAtMercury } }
}
CONCEPT_GOAL MMOStart_Orbit_Placement {
    SPEC {
       DEPART { MMO_Spacecraft.STATES.ArrivedAtMercury }
       ARRIVE { MMO_Spacecraft.STATES.InOrbitPlacement }
}}
```

The following code sample presents the specification of a goal with an arriving state expressed as a Boolean function over two MMO_Spacecraft states: *InHighIrradiation* and *AtPolarOrbit*.

```
CONCEPT_GOAL MMOSelf-Protection {
    SPEC {
       ARRIVE { NOT MMO_Spacecraft.STATES.InHighIrradiation AND
       MMO_Spacecraft.STATES.AtPolarOrbit} } }
```

In order to achieve specified goals (objectives), we need to specify *policies* triggering *actions* that will change the system states, so the desired ones, required by the goals, will become effective [6]. All the policies in KnowLang descend from the explicit *Policy* concept (see Figure 2). Note that policies allow the specification of *autonomic behavior* (autonomic behavior can be associated with autonomy requirements). As a rule, we need to specify at least one policy per single goal, i.e., a policy that will provide the necessary behavior to achieve that goal. Of course, we may specify multiple policies handling same goal (objective), which is often the case with the self-* objectives and let the system decides which policy to apply taking into consideration the current situation and conditions.

The following is a specification sample showing a simple policy called *BringMMOToOrbit* - as the name says, this policy is intended to bring MMO into polar orbit. As shown, the policy is specified to handle the goal *MMOOrbit_Placement_Done* and is triggered by the situation *ArrivedAtMercury*. Further, the policy triggers unconditionally (the *CONDITONS {}* directive is empty) the execution of the *GoToPolarOrbit* action.

```
CONCEPT_POLICY BringMMOToOrbit {
    SPEC {
       POLICY_GOAL { MMO..MMOOrbit_Placement_Done }
       POLICY_SITUATIONS { MMO..ArrivedAtMercury }
       POLICY_RELATIONS { MMO..Policy_Situation_2 }
       POLICY_ACTIONS { MMO..Action.GoToPolarOrbit }
       POLICY_MAPPINGS {
          MAPPING {
            CONDITIONS {}
            DO_ACTIONS { MMO..Action.GoToPolarOrbit } }
       }
} } }
```

The following specifies the *MMOProtect_spacecraft* policy intended to handle the *MMOSelf_Protection* objective with similar probability distribution. Probabilities are recomputed after every action execution, and thus the behavior change accordingly.

```
CONCEPT_POLICY MMOProtect_Spacecraft {
    SPEC {
        POLICY_GOAL { MMO..MMOSelf-Protection }
        POLICY_SITUATIONS { MMO..HighIrradiation }
        POLICY_RELATIONS { MMO..Policy_Situation_3 }
        POLICY_ACTIONS {
MMO..Action.CoverInstruments, MMO..Action.TurnOffElectronics,
MMO..Action.MoveSpacecraftUp, MMO..Action.MoveSpacecraftDown}
        POLICY_MAPPINGS {
            MAPPING {
                CONDITIONS { MMO..Metric.SolarRadiation.VALUE < 90 }
                DO_ACTIONS {
MMO..Action.ShadeInstruments, MMO..Action.TurnOffElectronics }
            }
            MAPPING {
                CONDITIONS { MMO..Metric.SolarRadiation.VALUE >= 90 }
                DO_ACTIONS { MMO..Action.MoveSpacecraftUp }
                PROBABILITY {0.5}
            }
            MAPPING {
                CONDITIONS { MMO..Metric.SolarRadiation.VALUE >= 90 }
                DO_ACTIONS { MMO..Action.MoveSpacecraftDown }
                PROBABILITY {0.4}
            }
            MAPPING {
                CONDITIONS { MMO..Metric.SolarRadiation.VALUE >= 90 }
                DO_ACTIONS {
                    GENERATE_NEXT_ACTIONS(MMO..MMO_Spacecraft) }
                PROBABILITY {0.1}
} } } }
```

As mentioned above, policies are triggered by situations. Therefore, while specifying policies handling system objectives, we need to think of important situations that may trigger those policies. A single policy requires to be associated with (related to) at least one situation, but for polices handling self-* objectives we eventually need more situations. Actually, because the *policy-situation relation* is bidirectional, it is maybe more accurate to say that a single situation may need more policies, those providing alternative behaviors. To increase the *goal-oriented autonomicity*, in this policy's specification, we used the special KnowLang operator *GENERATE_NEXT_ACTIONS*, which will automatically generate the most appropriate actions to be undertaken by the MMO spacecraft. The action generation is based on the computations performed by a special *reward function* implemented by the KnowLang Reasoner. The *KnowLang Reward Function* (KLRF) observes the outcome of the actions to compute the possible successor states of every possible action execution and grants the actions with special *reward number* considering the current system state (or states, if the current state is a composite state) and goals. KLRF is based on past experience and uses Discrete Time Markov Chains [11] for probability assessment after action executions.

Situations are specified with *states* and *possible actions*. To consider a situation effective (the system is currently in that situation), its associated states must be respectively effective (evaluated as true). For example, the situation *ArrivedAtMercury* is effective if the MMO Spacecraft state *ArrivedAtMercury* is effective.

```
CONCEPT_SITUATION ArrivedAtMercury {
   CHILDREN {}
   PARENTS {MMO..Situation}
    SPEC {
      SITUATION_STATES { MMO_Spacecraft.STATES.ArrivedAtMercury }
      SITUATION_ACTIONS { MMO..Action.GoToPolarOrbit,
          MMO..Action.WaitForInstructions, MMO..Action.ScheduleNewTask }
}}
```

The actions define what can be performed once the system falls in a particular situation. For example, the *ArrivedAtMercury* situation has three possible actions: *GoToPolarOrbit*, *WaitForInstructions*, *ScheduleNewTask*.

3.3.3. Monitoring. The *monitoring autonomy requirement* is handled via the explicit *Metric concept*. In general, a self-adaptive system has sensors that connect it to the world and eventually help it listen to its internal components. These sensors generate raw data that represent the physical characteristics of the world. In our approach, we assume that MMO's sensors are controlled by a software driver (e.g., implemented in C++) where appropriate methods are used to control a sensor and read data from it. By specifying a *Metric concept,* we introduce a class of sensors to the KB, and by specifying instances of that class, we represent the real sensors. KnowLang allows the specification of four types of metrics [6]:

- *RESOURCE* - measure resources like capacity;
- *QUALITY* - measure qualities like performance, response time, etc.;
- *ENVIRONMENT* - measure environment qualities and resources;
- *ENSEMBLE* - measure complex qualities and resources where the metric might be a function of multiple metrics.

The following is a specification of a metric used to assist in the specification of states and policy conditions.

```
CONCEPT_METRIC OutsideRadiation {
    SPEC {
        METRIC_TYPE { ENVIRONMENT }
        METRIC_SOURCE { RadiationMeasure.OutsideRadiation }
        DATA { DATA_TYPE { MMO..Sievert } VALUE { 1 } } } }
```

3.3.4. Awareness. The *awareness autonomy requirements* are handled by the KnowLang Reasoner (see Section 5.2. in D02-02, v.2.2). However, still we need to specify concepts and objects that will support the reasoner in its awareness capabilities. For example, we need to specify metrics that support both *self-* and *environment monitoring* (see Section 5.3.3). Next by specifying states where metrics are used we introduce awareness capabilities for *self-awareness* and *context-awareness*. Finally, with the specification of *situations* (see Section 5.3.2) we introduce the basis for *situational awareness*.

Other classes of awareness could draw attention to *specific states* and *situations*, such as *operational conditions* and *performance* (*operational awareness*), *control processes* (*control awareness*), *interaction processes* (*interaction awareness*), and *navigation processes* (*navigation awareness*).

3.3.5. Resilience, Robustness, Mobility, Dynamicity and Adaptability. *Resilience, robustness*, *mobility*, *dynamicity* and *adaptability* autonomy requirements might be handled by specifying special *soft goals*. For example, the requirement "*robustness: robust to communication losses*" and "*resilience: resilient to solar radiation*". These requirements can be specified as soft-goals leading the system towards "*reducing and copying with communication losses*" and "*preventing the MMO from taking self-protective actions if the radiation is relatively low"*. 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*. 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* and/or *policy conditions*.

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).

4. Conclusion

In this paper, we presented an Autonomy Requirements Engineering (ARE) approach intended to solve this problem. The proposed ARE model uses GORE approach to elicit and define the system goals, and then applies a special Generic Autonomy Requirements (GAR) model to derive and define assistive and often alternative goals (objectives) the system may pursue in the presence of factors threatening the achievement of the initial system goals. Once identified, the autonomy requirements might be further specified with a proper formal notation. This approach has been used in a joint project with ESA on identifying the autonomy requirements for the ESA's BepiColombo Mission. In this paper, we presented a case study where ARE was applied by putting GAR in the context of space missions to derive autonomy requirements and goals models incorporating autonomicity via self-* objectives.

Future work is mainly concerned with further development of the ARE model and further adaptation of KnowLang to validate autonomy requirements.

Acknowledgements

This work was supported by ESTEC ESA (contract No. 4000106016), by the European Union FP7 Integrated Project Autonomic Service-Component Ensembles (ASCENS), and by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre at University of Limerick, Ireland.

References

[1] E. Vassev and M. Hinchey, "The Challenge of Developing Autonomic Systems", *IEEE Computer*, IEEE Computer Society, 43 (12), 2010, pp. 93–96.

[2] A. Van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective", *Proceedings of the 22nd IEEE International Conference on Software Engineering (ICSE-00)*, ACM, 2000, pp. 5–19.

[3] R. Grard, M. Novara, and G. Scoon, *BepiColombo - A Multidisciplinary Mission to a Hot Planet*, ESA Bulletin, ESA, 103, 2000, pp. 11–19.

[4] E. Vassev and M. Hinchey, "On the Autonomy Requirements for Space Missions", *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing Workshops (ISCORCW 2013)*, IEEE Computer Society, 2013, to appear.

[5] E. Vassev and M. Hinchey, "Autonomy Requirements Engineering: A Case Study on the BepiColombo Mission", *Proceedings of the C* Conference on Computer Science & Software Engineering (C3S2E '13)*, ACM, 2013, to appear.

[6] E. Vassev and M. Hinchey, "Knowledge Representation and Reasoning for Self-adaptive Behavior and Awareness", *TCCI - Special Issue on ICECCS 2012*, Springer, 2013, pending.

[7] ESA, *BepiColombo Mercury Mission to be Launched in 2015*. Feb 28, 2012, url: http://sci.esa.int/science-e/www/object/index.cfm?fobjectid=50105

[8] M. Novara, "The BepiColombo ESA cornerstone mission to Mercury", *Acta Astronautica*, 51(1-9), 2002, pp. 387–395.

[9] H. Yamakawa et al., "Current Status of the BepiColombo/MMO Spacecraft Design", *Advances in Space Research*, 33(12), 2004, pp. 2133-2141.

[10] J. Benkhoff, "BepiColombo: Overview and Latest Updates", *European Planetary Science Congress, EPSC Abstracts*, 7, 2012.

[11] W. J. Ewens and G. R. Grant, "Stochastic Processes (i): Poison Processes and Markov Chains", *Statistical Methods in Bioinformatics*, 2nd ed., Springer, 2005.