EAI Endorsed Transactions on Self-Adaptive Systems Research Article ICST.ORG

Requirements Engineering for Self-Adaptive Systems with ARE and KnowLang

Emil Vassev[1]

[1]Lero–the Irish Software Engineering Research Center, University of Limerick, Limerick, Ireland

Abstract

This article presents an approach to Autonomy Requirements Engineering (ARE) that targets the integration and promotion of autonomy in software-intensive systems by providing a mechanism and methodology for elicitation and expression of autonomy requirements. ARE relies on *goal-oriented requirements engineering* to elicit and define system goals, and uses the *generic autonomy requirements* model to derive and define assistive and, eventually, alternative objectives. The system may pursue these *"self-* objectives"* in the presence of factors threatening the achievement of the initial system goals. Once identified, the autonomy requirements are specified with KnowLang, a formal language dedicated to knowledge representation and reasoning. To demonstrate both the ARE's and KnowLang's ability to handle autonomy requirements for self-adaptive systems, the approach is applied to Science Clouds, a self-adaptive cloud platform.

Received on 16 November 2014; accepted on 09 January 2015, published on 28 January 2015

Keywords: self-adaptive systems, requirements engineering, autonomy, ARE, KnowLang

Copyright © 2014 Emil Vassev, licensed to ICST. This is an open access article distributed under the terms of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/), which permits unlimited use, distribution and reproduction in any medium so long as the original work is properly cited.

doi:10.4108/sas.1.1.e6

1. Introduction

There are many advantages to self-adaptation. Among the most promising are the fact that self-adaptation enables software-intensive systems to become more versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changing operational contexts, environments or system characteristics. Although very promising, the paradigm brings a lot of challenges.

Among the many challenges software engineers must overcome are those related to elicitation and expression of autonomy requirements. Nowadays, requirements engineering for autonomous and self-adaptive systems appears to be a wide open research area with no definitive solution yet. The major problem is that the integration and promotion of autonomy in software-intensive systems is an extremely challenging task.

This article draws upon my experience with the Autonomy Requirements Engineering (ARE) [1, 2] approach to present its ability to handle autonomy requirements for self-adaptive systems. The ARE approach has been developed by Lero, the Irish Software Engineering Research Center, within the mandate of a joint project with ESA, the European Space Agency. The approach is intended to help engineers tackle the integration and promotion of autonomy in software-intensive systems. ARE combines special *generic autonomy requirements* (GAR) with *goal-oriented requirements engineering* (GORE) [3]. Using this approach, software engineers can determine what autonomic features to develop for a particular system as well as what artifacts that process might generate (e.g., goals models, requirements specification, etc.). To model and formalize the elicited requirements, ARE relies on KnowLang [4], a formal language dedicated to knowledge representation and reasoning for self-adaptive systems.

Both ARE and KnowLang have been used in a few projects to capture the autonomy requirements of self-adaptive systems. Some proof-of-concept examples are related to the ESA BepiColombo Mission [5–7] and the ASCENS Project's [8] case studies on Swarm Robotics [9], eMobility [10], and Science Clouds [11].

*Corresponding author. Email: emil.vassev@lero.ie

2. ARE – Autonomy Requirements Engineering

2.1. Understanding ARE

The first step in developing any new software-intensive system is to determine the system's functional and non-functional requirements. The former requirements define what the system will actually do, while the latter requirements refer to its qualities, such as performance, along with any constraints under which the system must operate. Despite differences in application domain and functionality, all autonomous systems extend upstream the regular software-intensive systems with special *self-managing objectives* (self-* objectives). Basically, the self-* objectives provide the system's ability to automatically discover, diagnose, and cope with various problems. This ability depends on the system's degree of *autonomicity*, *quality and quantity of knowledge*, *awareness and monitoring capabilities*, and quality characteristics such as *adaptability*, *dynamicity*, *robustness*, *resilience*, and *mobility*. Basically, this is the basis of the ARE approach [1, 2, 5–7]: autonomy requirements are detected as self-objectives backed up by different capabilities and quality characteristics outlined by the GAR model.

The ARE approach starts with the creation of a *goals model* that represents system objectives and their interrelationships. For this, we use GORE where ARE goals are generally modeled with intrinsic features such as *type*, *actor*, and *target*, with links to other goals and constraints in the requirements model. Goals models might be organized in different ways copying with the system specifics and engineers' understanding about the system goals. Thus, we may have 1) hierarchical structures where goals reside different levels of granularity; 2) concurrent structures where goals are considered as concurrent; etc. At this stage, the goals models are not formal and we use natural language along with UML-like diagrams to record them.

The next step in the ARE approach is to work on each one of the system goals along with the elicited environmental constraints to come up with the self-* objectives providing the autonomy requirements for this particular system's behavior. In this phase, we apply the GAR model to a system goal to derive autonomy requirements in the form of goal's supportive and alternative self-* objectives along with the necessary capabilities and quality characteristics. In the first part of this phase, we record the GAR model in natural language. In the second part though, we use a formal notation to express this model in a more precise way. Note that, this model carries more details about the autonomy requirements, and can be further used for different analysis activities, including requirements validation and verification.

2.2. System Goals and Goals Models

Goals have long been recognized to be essential components involved in the requirements engineering (RE) process [12]. To elicit system goals, typically, the system under consideration is analyzed in its organizational, operational and technical settings. Problems are pointed out and opportunities are identified. High-level goals are then identified and refined to address such problems and meet the opportunities. Requirements are then elaborated to meet those goals.

Goal identification is not necessarily an easy task [13–15]. Sometimes goals can be explicitly stated by stakeholders or in preliminary material available to requirements engineers. Often though, they are implicit so that goal elicitation has to be undertaken. The preliminary analysis of the current system along with the operational environment is an important source for goal identification. Such analysis usually results in a list of problems and deficiencies that can be formulated precisely. Negating those formulations yields a first list of goals to be achieved by the system-to-be. In my experience, goals can also be identified systematically by searching for intentional keywords in the preliminary documents provided. Once a preliminary set of goals and goal-related constraints is obtained and validated with stakeholders, many other goals can be identified by *refinement* and by *abstraction*, just by asking HOW and WHY questions about the goals/constraints already available [16]. Other goals are identified by resolving conflicts among goals or obstacles to goal achievement. Further, such goals might be eventually defined as self-* objectives.

Goals are generally modeled 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., main system objectives) might comprise related, low-level, sub-goals that can be organized to provide different alternatives of achieving the high-level goals. In ARE, goals are registered in plain text with characteristics like *actors*, *targets* and *rationale*. Moreover, inter-goal relationships are captured by *goals models* putting together all goals along with associated constraints. ARE's goals models are presented in UML-like diagrams. Goals models can assist us in capturing autonomy requirements in several ways [2, 5–7]:

1. An ARE goals model might provide the starting point for capturing autonomy requirements 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 to accomplish its goals.

2. ARE goals models might be used to provide a means to represent *alternative ways* where the objectives of the system can be met and analyze and rank these alternatives with respect to *quality concerns* and other constraints, e.g., environmental constraints:
   (a) This allows for *exploration and analysis of alternative system behaviors at design time*.
   (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) Not all the 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 alternatives at design time, we are minimizing the need for that difficult task.
3. ARE goals models might provide the traceability mechanism from design to requirements. When a change in requirements is detected at runtime, 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 affects a particular goal in the model, it is possible to see how this goal is decomposed and which parts of the system implementing the functionality needed to achieve that goal are in turn affected.
   (b) By analyzing a goals model, it is possible to identify how a failure to achieve some particular goal affects the overall objective of the system.
   (c) Highly variable goals 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. ARE goals models provide a unifying intentional view of the system by relating goals assigned to individual parts of the system (usually expressed as actors and targets of a goal) to high-level system objectives and quality concerns:
   (a) High-level objectives or quality concerns serve as the *common knowledge* shared among the autonomous system's parts (or components) to achieve the global system optimization. In this way, the system can avoid the pitfalls of missing the globally optimal configuration due to only relying on local optimizations.
   (b) Goals models might be used to identify part of the knowledge requirements, e.g., actors or targets.

Moreover, goals models might be used to manage conflicts among multiple goals including self-* objectives. Note that by resolving conflicts among goals or obstacles to goal achievement, new goals (or self-* objectives) may emerge.

2.3. Self-* Objectives and Autonomy-Assistive Requirements

Basically, the GAR (generic autonomy requirements) model follows the principle that despite their differences in terms of application domain and functionality, all autonomous systems are capable of autonomous behavior driven by one or more *self-management objectives* [2] that drive the development process of such systems. ARE uses goals models as a basis helping to derive self-* objectives per a system goal by applying a model for *generic autonomy requirements* to any system goal [2, 6]. The self-* objectives represent assistive and eventually *alternative goals* (or objectives) the system may pursue in the presence of factors threatening the achievement of the initial system goals. The diagram presented in Figure 1 depicts the process of deriving the self-* objectives from a goals model of the system-to-be. Basically, a context-specific GAR model provides some initial self-* objectives, which should be further analyzed and refined in the context of the specific system goal to see their applicability. As shown in Figure 1, in addition to the derived self-* objectives, the ARE process also produces *autonomy assistive requirements*. These requirements (also defined as adaptation-assistive attributes) are initially defined by the GAR model [1, 2, 5] and are intended to support the achievements of the self-* objectives. The autonomy assistive requirements outlined by GAR might be defined as following:

- *Knowledge* - basically data requirements that need to be structured to allow efficient reasoning.
- *Awareness* - a sort of functional requirements where knowledge is used as an input along with

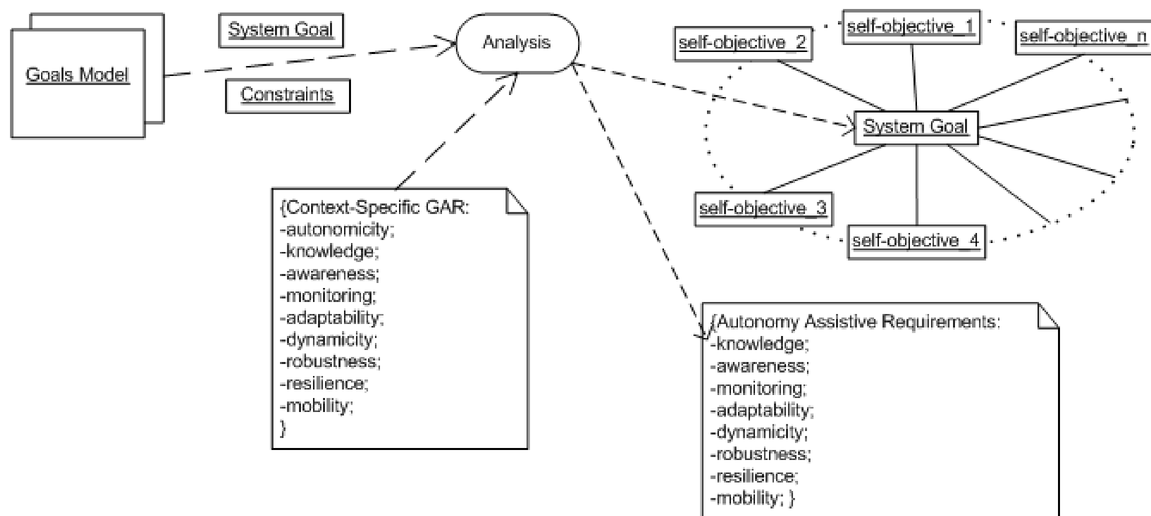

Figure 1. The ARE Process of Deriving Self-* Objectives per System Goal

events and/or sensor signals to derive particular system states.

- *Resilience* and *robustness* - a sort of soft-goals. For example, such requirements can be defined as "*robustness: system is robust to communication latency*" and "resilience: system is resilient to hardware failures, node disappearances, or appearances". These requirements can be specified as soft goals leading the system towards "*reducing and copying with communication latency*" and "*keeping system's performance optimal*". A soft goal is satisfied rather than achieved. 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 satisfied 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.

- *Monitoring*, *mobility*, *dynamicity* and *adaptability* - might also be defined 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 needs 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).

2.4. Autonomy Needs and Requirements Chunks

To record autonomy requirements, ARE relies on both natural language and formal notation. A natural language description of a self-* objective has the following format [6]:

- **Name of Self-* Objective**: *Rationale of this self-* objective.*
  - **Assisting system goals**: *List of system goals assisted by this self-* objective.*
  - **Actors**: *Actors participating in the realization of this self-* objective.*
  - **Targets**: *Targets of this self-* objective.*

Note that this description is abstract and does not say how the self-* objective is going to be achieved. Basically, as recorded the self-* objectives define the "*autonomy needs*" of the system. How these needs are going to be met is provided by more detailed description of the self-* objectives recorded as ARE Requirements Chunks and/or specified formally.

In general, a more detailed description in a natural language may precede the formal specification of the elicited autonomy requirements. Such description might be written as a scenario describing both the conditions and sequence of actions needed to be performed in order to achieve the self-* objective in question.

Note that a self-objective could be associated with multiple scenarios. The combination of a self-* objective and a scenario forms an *ARE Requirements Chunk* (see Figure 2). A requirements chunk can be recorded in a natural language as following:

ARE Requirements Chunk

- **Name of Self-* Objective**: *Rationale of this self-* objective.*
  - **Assisting system goals**: *List of system goals assisted by this self-* objective.*
  - **Actors**: *Actors participating in the realization of this self-* objective.*
  - **Targets**: *Targets of this self-* objective.*
- **Scenario**: *Description of a scenario how this self-* objective can be met by performing the system's functionality.*

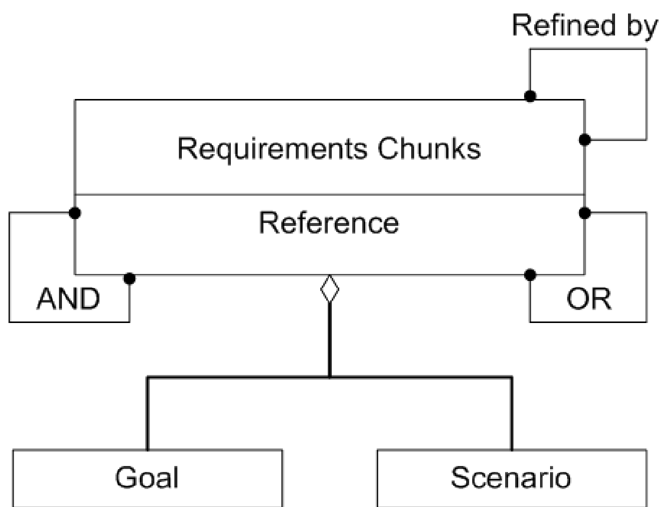

Figure 2. Requirements Chunk – Goal & Scenario

Requirements chunks associate each goal with scenarios where the *goal-scenario pairs* can be assembled together through *composition*, *alternative* and *refinement relationships* (see Figure 2). The first two lead to AND and OR structures of requirements chunks, whereas the last leads to the organization of the collection of requirements chunks as a hierarchy of chunks of different granularity. *AND relationships* among requirements chunks link complementary chunks in the sense that everyone requires others to define a completely functioning scenario covering a main goal. Requirements chunks linked through *OR relationships* represent alternative ways of fulfilling the same goal. Requirements chunks linked through a *refinement relationship* are at different levels of abstraction. Internally, the scenarios might introduce additional variability via *conditional requirements* derived from the GAR's requirements such as *monitoring*, *adaptability*, *dynamicity*, *resilience*, and *robustness*.

2.5. Formal Specification

ARE relies on KnowLang [4] for the formal specification of the elicited autonomy requirements. We use KnowLang to record these requirements as knowledge representation in a Knowledge Base (KB) comprising a variety of knowledge structures, e.g., *ontologies*, *facts*, *rules*, and *constraints*. The self-* objectives are specified with special *policies* associated with *goals*, special *situations*, *actions* (eventually identified as system capabilities), *metrics*, etc. Thus, the self-* objectives are represented as policies describing at an abstract level what the system will do when particular situations arise. The situations are meant to represent the conditions needed to be met in order for the system to switch to a self-* objective while pursuing a system goal. Note that the policies rely on actions that are a priori-defined as functions of the system. In case, such functions have not been defined yet, the needed functions should be considered as *autonomous functions* and their implementation will be justified by the ARE's selected self-* objectives. ARE does not state neither specify how the system will perform these actions. This is out of the scope of the ARE approach. Basically, any requirements engineering approach states what the software will do not how the software will do it.

3. KnowLang

A key feature of KnowLang [4] is a formal language with a multi-tier knowledge specification model allowing for integration of ontologies together with rules and Bayesian networks [17]. The language aims at efficient and comprehensive knowledge structuring and awareness based on logical and statistical reasoning. It helps us tackle: 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 [18]. Other noteworthy features are related to knowledge cleaning (allowing for efficient reasoning) [4, 18] and knowledge representation for autonomic behavior [4, 19]. By applying the KnowLang's multi-tier specification model (see Figure 3) we build a Knowledge Base (KB) structured in three main tiers [4, 18]: 1) *Knowledge Corpuses*; 2) *KB Operators*; and 3) *Inference Primitives*. The tier of Knowledge Corpuses is used to specify knowledge representation (KR) structures. The tier of KB Operators provides access to Knowledge Corpuses via special classes of *ASK* and *TELL Operators*, where ASK Operators are dedicated to knowledge querying and retrieval and TELL Operators allow for knowledge update. When we specify knowledge with KnowLang, we build a KB with a variety of knowledge structures such as

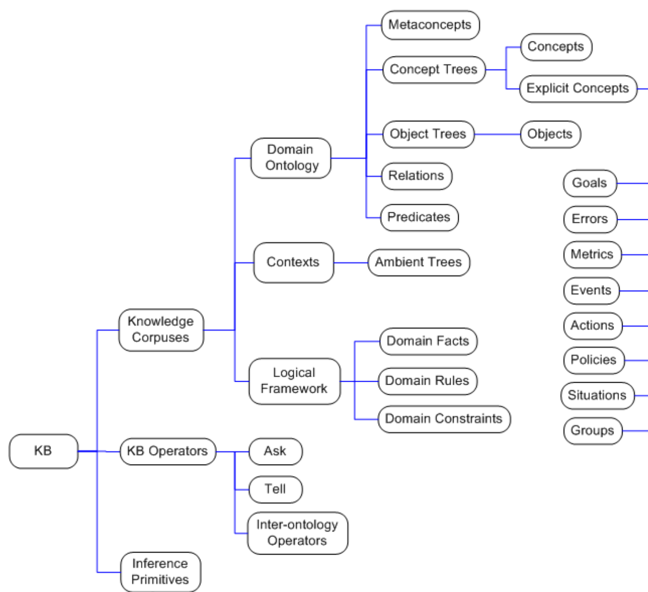

Figure 3. KnowLang Specification Model

ontologies, *facts*, *rules* and *constraints* where we need to specify the ontologies first in order to provide the "vocabulary" for the other knowledge structures. A KnowLang ontology is specified over *concept trees*, *object trees*, *relations* and *predicates*. Each concept is specified with special properties and functionalities and is hierarchically linked to other concepts through *PARENTS* and *CHILDREN* relationships. For reasoning purposes every concept specified with KnowLang has an intrinsic *STATE* attribute that may be associated with a set of possible *state values* these concept instances may be in. Concept instances are considered as objects and are structured in object trees — 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. Moreover, concepts and objects may be connected via *relations*. Relations are binary and may have probability-distribution attributes (e.g., over time, over situations, over concepts' properties, etc.). Probability distribution is provided 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 4 shows a KnowLang specification sample demonstrating both the language syntax [20] and its visual counterpart — a concept map based on interrelations with no probability distributions. Modeling knowledge with KnowLang requires a number of phases:

- Initial knowledge gathering – involves domain experts to determine the basic notions, relations and functions (operations) of the domain of interest.
- Behavior definition – identifies situations and behavior policies as "control data" helping to identify important self-adaptive scenarios.
- Knowledge structuring – encapsulates domain entities, situations and behavior into KnowLang structures such as concepts, objects, relations, facts and rules.

3.1. Modeling Self-adaptive Behavior

KnowLang employs special knowledge structures and a reasoning mechanism for modeling autonomic self-adaptive behavior [19]. Such a behavior can be expressed via KnowLang *policies*, *events*, *actions*, *situations* and *relations* between policies and situations (see Definitions 1 through 10). Policies (Π) are at the core of autonomic behavior. 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 $N_\pi \xrightarrow{[Z]} A_\pi$) (see Definition 2). A condition is a Boolean expression over an ontology (see Definition 4), e.g., the occurrence of a certain event.

Policy situations Si_π are situations (see Definition 7) that may trigger (or imply) a policy π, in compliance with the policy-situations relations R_π(denoted by $Si_\pi \xrightarrow{[R_\pi]} \pi$), thus implying the evaluation of the policy conditions N_π(denoted by $\pi \rightarrow N_\pi$)(see Definition 2). Therefore, the optional policy-situation relations (R_π) justify the relationships between a policy and the associated situations (see Definition 10). Note that in order to allow for self-adaptive behavior, *relations* must 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 reasoner to 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 reasoner decide which policy to choose (denoted by $si \xrightarrow{[Z]} \pi$ – see Definition 10). Hence, the presence of *probabilistic beliefs* in both mappings and policy relations justifies the probability of policy execution, which may vary with time. A goal g is a desirable transition to a state, or from a specific state to another state, (denoted by $s \Rightarrow s'$) (see Definition 5). A state s is a Boolean expression over ontology ($be(O)$)(see Definition 6), e.g., "a specific property of an object must hold a specific value". A situation is

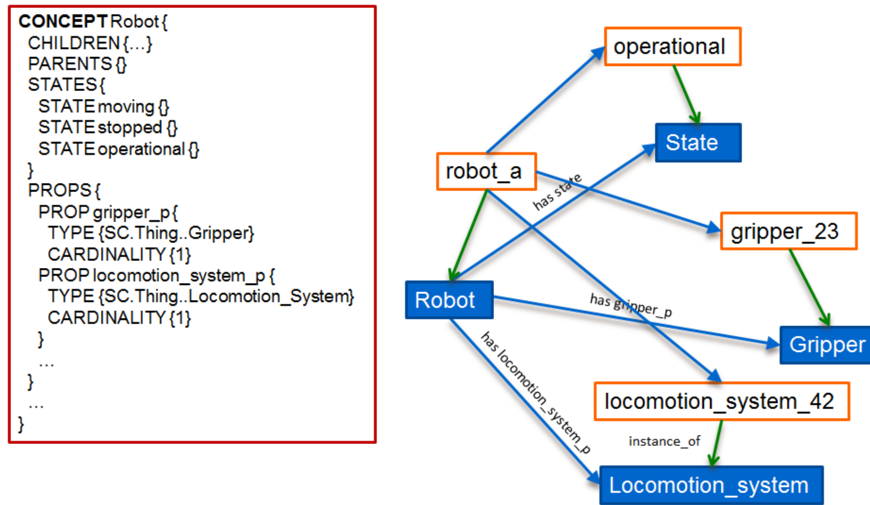

Figure 4. KnowLang Specification Sample

expressed with a state (s), a history of actions ($A_{\overleftarrow{si}}$) (actions executed to get to state s), actions A_{si} that can be performed from state s and an optional history of events $E_{\overleftarrow{si}}$ that eventually occurred to get to state s (see Definition 8).

Def. 1. $\Pi := \{\pi_1, \pi_2, ..., \pi_n\}, n \geq 0$ (Policies)

Def. 2. $\pi :=< g, Si_\pi, [R_\pi], N_\pi, A_\pi, map(N_\pi, A_\pi, [Z]) >$

$A_\pi \subset A, N_\pi \overset{[Z]}{\to} A_\pi$ (A_π - Policy Actions)

$Si_\pi \subset Si, Si_\pi \overset{[R_\pi]}{\to} \pi \to N_\pi$ (Si_π - Policy Sitns)

$R_\pi \subset R$ (R_π-Policy-Situation Relations)

Def. 3. $N_\pi := \{n_1, n_2, ..., n_k\}, k \geq 0$ (Policy Condtns)

Def. 4. $n := be(O)$ (Boolean Expression over Ontology)

Def. 5. $g := \langle \Rightarrow s' \rangle | \langle s \Rightarrow s' \rangle$ (Goal)

Def. 6. $s := be(O)$ (State)

Def. 7. $Si := \{si_1, si_2, ..., si_n\}, n \geq 0$ (Situations)

Def. 8. $si :=< s, A_{\overleftarrow{si}}, [E_{\overleftarrow{si}}], A_{si} >$ (Situation)

$A_{\overleftarrow{si}} \subset A$ ($A_{\overleftarrow{si}}$ - Executed Actions)

$A_{si} \subset A$ (A_{si} - Possible Actions)

$E_{\overleftarrow{si}} \subset E$ ($E_{\overleftarrow{si}}$ - Situation Events)

Def. 9. $R := \{r_1, r_2, ..., r_n\}, n \geq 0$ (Relations)

Def. 10. $r :=< \pi, [rn], [Z], si >$ (rn - Relation Name)

$si \in Si, \pi \in \Pi, si \overset{[Z]}{\to} \pi$

Ideally, KnowLang 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 – see Definition 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 ($map(N_\pi, A_\pi, [Z])$) – see Definition 2). An optional probability distribution may additionally restrict the action execution. Although specified initially, 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*.

3.2. Converting Sensory Data to KR

One of the biggest challenges is *"how to map sensory raw data to KR symbols"*. My approach to this problem is to specify special explicit concepts called *METRICS*. In general, a self-adaptive system has sensors that connect it to the real world and eventually help it listen to its internal components. These sensors generate raw data that represent the physical characteristics of the world. The problem is that these low-level data streams must be: 1) converted to programming variables or more complex data structures that represent collections of sensory data; 2) those programing data structures must be labeled with KR symbols. Hence, it is required to relate encoded data structures with KR concepts and objects used for reasoning purposes. In this approach, we assume that each sensor is controlled by a software driver (e.g., implemented in Java) where appropriate

methods are used to control the sensor and read data from it. Both the *sensory data* and *sensors* should be represented in the KB by using *METRIC* explicit concepts and instantiate objects of these concepts. By specifying a METRIC concept we introduce a *class of sensors* to the KB and by specifying objects, instances of that class, we give the actual KR of a real sensor. KnowLang allows the specification of four different types of metrics [20]:

- 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; might be a function of multiple metrics both of RESOURCE and QUALITY type.

3.3. KnowLang Reasoner

A very challenging task is the R&D of the inference mechanism providing for *knowledge reasoning and awareness*. In order to support reasoning about self-adaptive behavior and to provide a KR gateway for communication with the KB, a special KnowLang Reasoner has been developed. The reasoner communicates with the system and operates in the KR Context, a context formed by the represented knowledge (see Figure 5).

The KnowLang Reasoner should be supplied as a component hosted by the system and, thus, it runs in the system's Operational Context as any other system's component. However, it operates in the 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 (See Figure 5). 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.

4. Capturing Autonomy Requirements for Science Clouds

To better understand the concepts behind ARE, in this section, is presented an example of using the ARE approach to capture autonomy requirements for an autonomic system described as Science Clouds.

4.1. Science Clouds

Science Clouds is a cloud computing scientific platform for application execution and data storage [21], tackled by the ASCENS Project [8] as a case study. Individual users or universities can join a cloud to provide (and consume of course) resources to the community. A science cloud is a collection of cloud machines - notebooks, desktops, servers, or virtual machines, running the Science Cloud Platform (SCP). Each machine is usually running one instance of the Science Cloud Platform (Science Cloud Platform instance or SCPi). Each SCPi is considered to be a Service Component (SC) in the ASCENS sense . To form a cloud, multiple SCPis communicate over the Internet by using the IP protocol. Within a cloud, a few SCPis might be grouped into a Service Component Ensemble (SCE), also called a Science Cloud Platform ensemble (SCPe). The relationships between the SCPis are dynamic and the formation of a SCPe depends mainly on the properties of the SCPis. The common characteristic of an ensemble is SCPis working together to run one application in a fail-safe manner and under consideration of the Service Level Agreement (SLA) of that application, which may require a certain number of active SCPis, certain latency between the parts, or have restrictions on processing power or memory. The SCP is a *platform as a service* (PaaS), which provides a platform for application execution [22]. Thus, SCP provides an execution environment where special applications might be run by using the SCP's application programming interface (API) and SCP's library [22]. These applications provide a *software as a service* (SaaS) cloud solution to users. The data storage service is provided in the same manner, i.e., via an application.

Based on the rationale above, we may conclude that the Science Clouds' main objective is to *provide a scientific platform for application execution and data storage* [21]. Being a cloud computing approach, the Science Clouds approach extends the original cloud computing goal to *provide services* (or resources) to the community of users. Note that cloud computing targets three main types of service (or resource):

1. Infrastructure as a Service (IaaS): a solution providing resources such as virtual machines, network switches and data storage along with tools and APIs for management (e.g., starting VMs).
2. Platform as a Service (PaaS): a solution providing development and execution platforms for cloud applications.
3. Software as a Service (SaaS): a solution providing software applications as a resource.

4.2. GORE for Science Clouds

The three different services provided by Science Clouds (see Section 4.1) can be defined as three main goals of cloud computing, and their realization by Science Clouds will define the main Science Clouds goals.

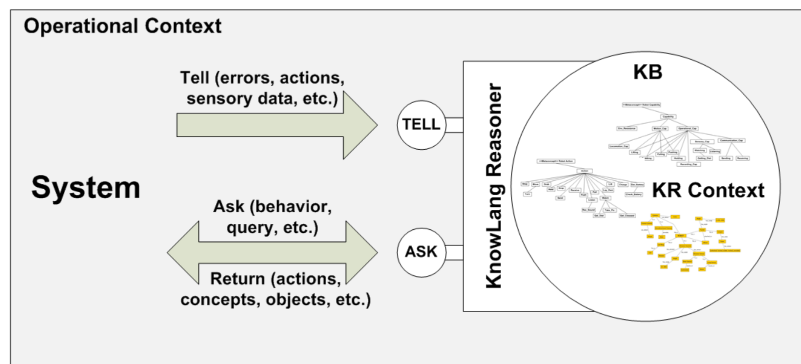

Figure 5. KnowLang Reasoner

Figure 6 depicts the ARE goals model for Science Clouds where goals are organized hierarchically at four different levels. In addition, from the rationale above we may conclude that an underlying system goal is to optimize application execution by minimizing resource usage along with providing a fail-safe execution environment.

As shown in Figure 6, the goals from the first three levels are main system goals captured at different levels of abstraction. The 3rd level is resided by goals directly associated with Science Clouds and providing a concrete realization of the cloud computing goals outlined at the first two levels. Finally, the goals from the 4th level are supporting and preliminary goals that need to be achieved before proceeding with the goals from the 3rd level. Figure 6 puts together all the system goals by relating them via particular relationships such as inheritance and dependency. Goals are depicted as boxes listing both goal actors and targets (note that targets might be considered as a distinct class of actors). The ARE Goals Model for Science Clouds provides the traceability mechanism for autonomy requirements. When a change in requirements is detected at runtime, the goals model can be used to re-evaluate the system behavior with respect to the new requirements and to determine if system reconfiguration is needed. Moreover, the presented goals model provides a unifying intentional view of the system by relating goals assigned to actors and involving targets. Some of the actors can be eventually identified as the autonomy components providing a self-adaptive behavior when necessary to keep up with the high-level system objectives (the goals residing Level 3).

The following elements describe the system goals by goal levels as shown in Figure 6:
Level 1 Goals:

- **Provide Resources**: *A cloud computing system (cloud) shall provide computational resources to the community of users.*
  - **Actors**: *cloud (the cloud computing system), users*
  - **Targets**: *resources*

Level 2 Goals:

- **Provide Infrastructure as a Service**: The cloud shall provide resources such as virtual machines, virtual network switches, and data storage. To manage this infrastructure, the cloud provides tools and APIs for management, e.g., starting and stopping VMs or creating new virtual networks.
  - **Actors**: cloud, operators
  - **Targets**: virtual machines, network switches, data storage

- **Provide Platform as a Service**: The cloud shall provide development and execution platforms for cloud applications, e.g., it may provide a framework for writing applications (by developers), which can either be supplied with adequate resources and distributed automatically, or request additional resources.
  - **Actors**: cloud, developers
  - **Targets**: development platforms, execution platforms

- **Software as a Service**: The cloud shall provide software applications that can be run by users within the cloud. Some examples of such applications could be e-mail service, word processor, etc. A good real-life example is Google Apps.
  - **Actors**: cloud, execution platform, users
  - **Targets**: applications platforms

Level 3 Goals:

- **Provide Zimory Cloud**: This goal is to realize the Provide Infrastructure as a Service cloud computing goal by running the Zimory Cloud.

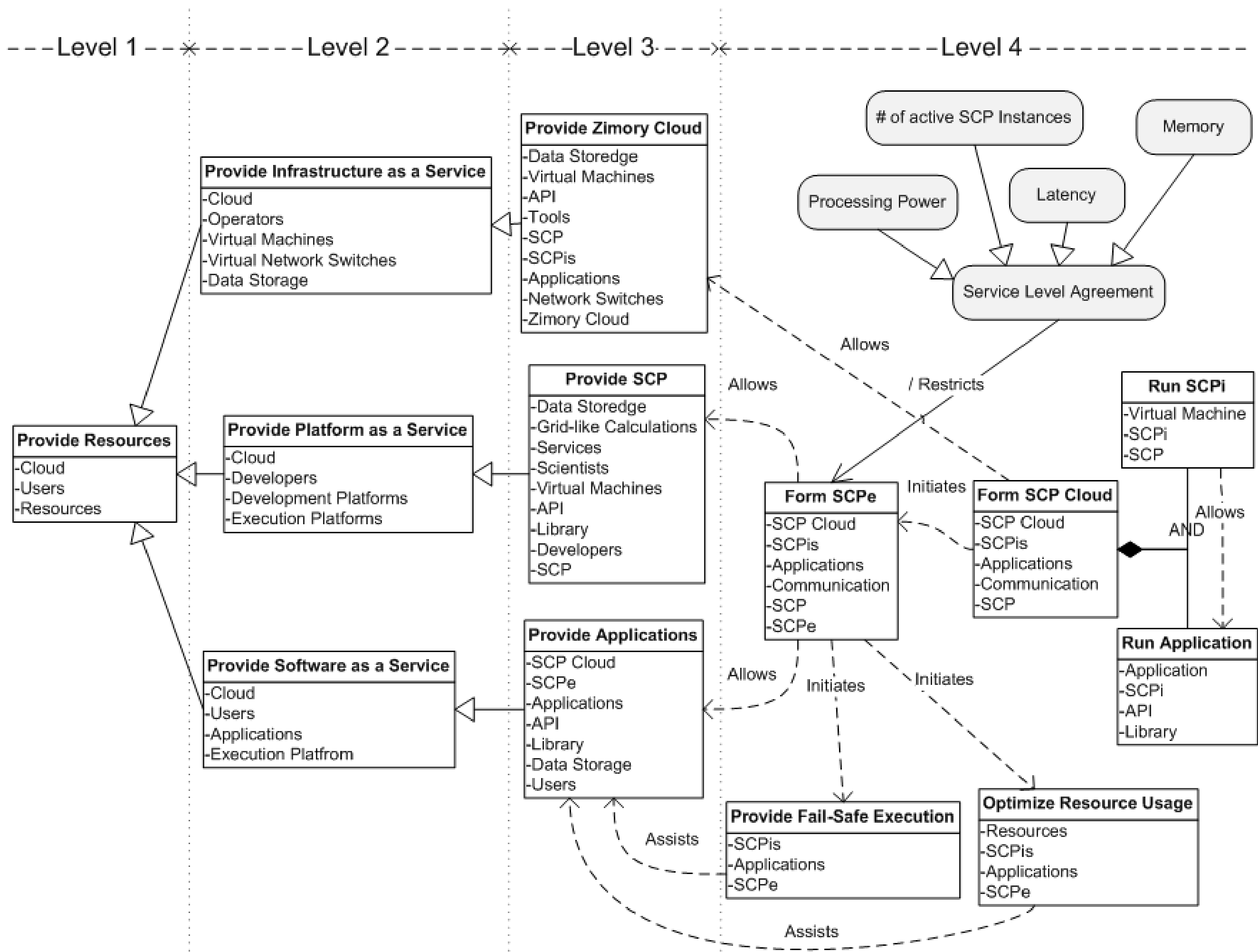

Figure 6. Science Clouds Goals Model

The Zimory Cloud shall provide cloud infrastructure based on SCP by running SCPis on virtual machines, as described by the rationale above. In addition, the goal requires that the Zimoty Cloud provide both API and tools needed for infrastructure management.

- **Actors**: Zimory Cloud, API, tools, SCP, SCPis, operators
- **Targets**: virtual machines, network switches, data storage, applications

• **Provide SCP**: This goal is to realize the Provide Platform as a Service cloud computing goal by providing the Zimory Cloud's SCP. The SCP must ensure both development and execution platforms where cloud applications can be developed and executed. Therefore, the platform must provide both API and libraries used by developers.

- **Actors**: SCP, developers, scientists
- **Targets**: API, library, virtual machines, services, grid-like calculations, data storage

• **Provide Applications**: This goal is to realize the Provide Software as a Service cloud computing goal by providing applications running in the SCP Cloud (or Zimory Cloud). The software applications can be run within a SCPe by users using the SCP's application programming interface (API) and SCP's library. Data storage services might be provided via applications as well.

- **Actors**: SCP Cloud, SCPe, API, library, users
- **Targets**: applications, data storage

Level 4 Goals:

• **Form SCPe**: This goal is to form a dynamic SCPe that shall provide the needed computational resources for the realization of either the Provide SCP goal or Provide Applications goal, or both.

The Form SCPe goal is supportive to these two goals (see the *allows* relationship in Figure 6). Moreover, the achievement of this goal may initiate two more assistive goals: Provide Fail-safe Execution and Optimize Resource Usage, which assist the Provide Applications goal (see Figure 6). Note that this goal shall take into consideration the Service Level Agreement constraint, which may impose restrictions (or requirements) on the processing power, number of SCPis running within the ensemble, communication latency, memory usage, etc.

  – **Actors**: SCP Cloud, SCPis, application, communication, Service Level Agreement
  – **Targets**: SCPe

- **Form SCP Cloud**: This goal is to form the SCP Cloud (Zymory Cloud) from the running SCPis joining their resources within that cloud. Note that the cloud allows the individual SCPis voluntarily join in or opt out. In addition, any application that runs on a cloud's SCPi is also added to the cloud as a resource. Thus, the SCP Cloud is formed by both running SCPis and applications (see Figure 6).
  – **Actors**: SCP, SCPis, application, communication
  – **Targets**: SCP Cloud
- **Run SCPi**: This goal is to run a SCPi as an instance of SCP hosted by a virtual machine. Basically, this goal along with the Run Application goal (both connected via AND relationship) might be considered as a sub-goal of the Form SCP Cloud goal.
  – **Actors**: SCP, virtual machine
  – **Targets**: SCPi
- **Run Application**: This goal is to run an application on a SCPi using SCP's API and library. This goal must be achieved as part of the Form SCP Cloud goal, i.e., it might be considered as a sub-goal of this goal.
  – **Actors**: SCPi, API, library
  – **Targets**: application
- **Provide Fail-safe Execution**: This goal is to ensure that running applications will continue working if a hosting SCPi fails. This policy must be provided by a SCPe, eventually formed to provide a fail-safe execution environment. The Provide Fail-safe Execution goal is assistive to the Run Application goal and it may be considered as a self-* objective providing fault tolerance.
  – **Actors**: applications, SCPis, SCPe
  – **Targets**: fail-safe execution of applications
- **Optimize Resource Usage**: This goal is to ensure that running applications will use the cloud resources in the most optimal way. This policy must be provided by a SCPe, eventually formed to provide an optimal use of particular cloud resources, e.g., memory, disk space, etc. The Optimize Resource Usage goal is assistive to the Run Application goal and it may be considered as a self-* objective providing self-optimization.
  – **Actors**: applications, SCPis, SCPe, cloud resources
  – **Targets**: optimized resource usage

4.3. GAR for Science Clouds

After completing the goals model for Science Clouds, the next step of the ARE approach is to put the GAR model in the context of cloud computing to derive a domain-specific GAR that can be applied to the goals captured by the goals model for Science Clouds. To derive the domain-specific GAR, we need to elaborate on the Science Clouds features, issues and goals to come up with self-* objectives and the consecutive autonomy-assistive requirements. For example, some remarkable issues that eventually can turn to autonomy features are [21]:

- *fail-safe operation*: An application should be available even its host SCPi fails (see Provide Fail-safe Execution goal in Section 4.2).
- *load balancing / throughput*: Parallel execution of same applications to distribute the computational/resource overhead (load) when it is high, but not before that.
- *energy conservation*: Shutting down virtual machines or de-configuring virtual networks if not required (this feature requires IaaS support).
- *SCPi fails, disappears, or appears*: A failing SCPi attempts to notify other SCPis, which need to take over responsibilities. If a new SCPi appears, it should engage with applications execution.
- *SCPi (or link) with high load, or idle*: Move applications to another SCPi, receive applications from another SCPi, or run a new SCPi on a virtual machine. If a SCPi is idle, then engage with applications running already on another SCPi, or simply shut down it.

To address these issues, SCPis must be monitored (including self-monitored) along with the cloud environment to detect high computational loads (due to applications), high communication latency, high memory usage, other SCPis that join in or opt out, etc. Basically, monitoring shall go on three levels:

- *network level*: The SCPis forming a SCPe need to know each other and be able to route between themselves.
- *application level*: The SCPis forming a SCPe need to know what applications run on which SCPis.
- *data level*: When an application is deployed, the SCPis that can eventually run that application need to have the application executable (immutable data). Moreover, the SCPis running that application need to monitor the application data (mutable data) and eventually store it through check points, so the application can be resumed in case of a SCPi failure or the failure of the application itself.

Addressing these issues in the context of the system goals (see Section 4.2) will result into self-adaptive behavior realized by self-* objectives. These self-* objectives along with the autonomy-assistive requirements form the domain-specific GAR model for Science Clouds as following:

- *self-* objectives (autonomicity)*:
  - *self-healing*: If a SCPi fails or is shut down, the applications executing on it must be made available on another SCPi in the SCPe hosting those SCPis.
  - *self-configuring_1*: Each SCPi is aware about changes in its hosting SCPe - new SCPis can be added to the hosting SCPe or other can voluntarily leave of shut down. A SCPi should adapt itself to take into consideration both the newly available resources and recently disappeared resources provided by other SCPis.
  - *self-configuring_2*: A SCPi is aware about the performance of the hosted applications. If an application is slowing down due to a lack of resources, this application can be distributed among different SCPis (run/resumed in parallel) if the application itself supports distributed execution.
  - *self-optimizing_1*: If a SCPi reaches its capacity (e.g., consistent high CPU load or swapping due to high memory usage), it may transfer some of the computational load to another SCPi from the same SCPe.
  - *self-optimizing_2*: If the communication latency within a SCPe is relatively high, due to overloaded links in the network, the SCPe may engage new SCPis to reduce the communication traffic.
  - *self-optimizing_3*: If the communication latency within a SCPe is relatively high, due to overloaded links in the network, the SCPe may reduce the load transfer within the SCPe itself.
  - *self-optimizing_4*: If SCPis are no longer required, the hosting SCPe may reconfigure to engage the idle SCPis in computational processes.
  - *self-optimizing_5*: If certain SCPis are no longer required, they may shut down along with their hosting virtual machines to save energy.
  - *self-optimizing_6*: If the computational load in certain SCPes is relatively high, due to overloaded application executions, the SCPe may start new SCPis along with the hosting virtual machines (if necessary) to reduce the computational overload.
- *knowledge*: cloud objectives; SCPes (engaged SCPis, ensemble's applications, ensemble's virtual machines, service level agreement, states), SCPis (applications, CPU, memory, storage capacity, states); applications (needed resources, distributiveness, states); communication links;
- *awareness*: application awareness (resource consumption, execution stage, load distribution, data-transfer); SCPi self-awareness (applications, resources, hosting virtual machine, user); SCPe awareness (participating SCPis, communication links, distributed applications, service level agreement); cloud awareness (SCPes, SCPis); communication awareness (communicating SCPis, data-transfer);
- *monitoring*: SCPi self-monitoring (running applications, CPU load, memory usage, storage capacity); SCPe monitoring (ensemble's SCPis, communication latency between SCPis, data transfer within SCPe);
- *adaptability*: adaptable load balancing; adaptable communication;
- *dynamicity*: dynamic communication links; dynamic SCPe formation;
- *robustness*: robust to SCPi failures; robust to data-transfer failures; robust to application execution failures;

- *resilience*: resilient communication links (communication losses must be repairable); network resilience (the routing needs to work in a dynamic environment where SCPis voluntarily join in and opt out of SCPes); application resilience; data resilience;
- *mobility*: data distribution; application distribution; SCPi mobility (SCPis may run on different virtual machines);

4.4. ARE Requirements Chunks for Science Clouds

The next step is to *merge* the GORE model for Science Clouds with the GAR model for science clouds, by applying the GAR model to the system goals captured in the first phase of the ARE process. Considering the fact that the Level 3 goals (see Figure 6 and Section 4.2) present the main system goals, we can apply the GAR model to these goals to derive self-adaptive behavior supporting the common Science Clouds behavior realized by the goals *Provide Zimory Cloud*, *Provide SCP*, and *Provide Applications*. Note that not all the self-* objectives derived by the GAR model in Section 4.3 are relevant to every one of these three goals. In this section is presented the self-* objectives derived for these three goals. The self-* objectives are presented as autonomy requirements chunks (see Section 4.5).

For the *Provide Zimory Cloud* goal, are derived the following self-* objectives:

- **Self-Optimizing_5**: If certain SCPis are no longer required, they may shut down along with their hosting virtual machines to save energy.
  - **Assisting system goals**: Provide Zimory Cloud
  - **Actors**: SCPis, virtual machines
  - **Targets**: SCPis shut down
  - **Scenario**: If a SCPi is in idle mode during a certain interval of time, then it can autonomously shut down. If a hosting virtual machine detects that it is not running any SCPis for a certain period of time, it can autonomously shut down.
- **Self-Optimizing_6**: If the computational load in a SCPe is relatively high, due to overloaded application executions, the SCPe may start new SCPis along with the hosting virtual machines (if necessary) to reduce the computational overload.
  - **Assisting system goals**: Provide Zimory Cloud
  - **Actors**: SCPe, SCPis, virtual machines, applications
  - **Targets**: SCPis started,
  - **Scenario**: If a SCPe detects a high computational load in the entire ensemble of SCPis, i.e., all the engaged SCPis run heavy application executions, then it may start new SCPis. If there is a lack of virtual machines that can host SCPis, then such machines can be started as well.

For the *Provide SCP* goal, are derived the following self-* objectives:

- **Self-Configuring_1**: Each SCPi is aware about changes in its hosting SCPe - new SCPis can be added to the hosting SCPe or other can voluntarily leave of shut down. A SCPi should adapt itself to take into consideration both the newly available resources and recently disappeared resources provided by other SCPis.
  - **Assisting system goals**: Provide SCP
  - **Actors**: SCPe, SCPis, applications
  - **Targets**: SCPis updated on changes in resource availability
  - **Scenario**: If a SCPi detects absence of a previously active SCPi it stops collaborating with that SCPi, i.e., it stops all the joint operations on applications execution and data transferring. Moreover, the active SCPi may need to reconsider the resource availability and eventually reschedule the controllable application executions to cope with the new situation. If a SCPi detects presence of a new SCPi that recently joined the SCPe, it shall reconsider the resource availability and eventually it may ask this new SCPi share part of the computational workload.
- **Self-optimizing_1**: If a SCPi reaches its capacity (e.g., consistent high CPU load or swapping due to high memory usage), it may transfer some of the computational load to another SCPi from the same SCPe.
  - **Assisting system goals**: Provide SCP
  - **Actors**: SCPe, SCPis, resources, applications
  - **Targets**: application executions shared among SCPis
  - **Scenario**: If a SCPi detects high resource usage (consistent high CPU load or high swapping) it may ask another SCPi to take over some of the application executions.
- **Self-optimizing_2**: If the communication latency within a SCPe is relatively high, due to overloaded

links in the network, the SCPe may engage new SCPis to reduce the communication traffic.

- **Assisting system goals**: Provide SCP
- **Actors**: SCPe, SCPis, communication
- **Targets**: low communication latency
- **Scenario**: If a SCPi detects high communication latency while communicating with another SCPi, it may start collaborating with other SCPis to reduce the data transfer with the initial SCPi and consecutively, reduce the communication latency.

• **Self-optimizing_3**: If the communication latency within a SCPe is relatively high, due to overloaded links in the network, the SCPe may reduce the load transfer within the SCPe itself.

- **Assisting system goals**: Provide SCP
- **Actors**: SCPe, SCPis, communication, transferred data
- **Targets**: low communication latency
- **Scenario**: If a SCPi detects high communication latency while communicating with another SCPi, it may reduce the amount of transferred data.

• **Self-Optimizing_4**: If SCPis are no longer required, the hosting SCPe may reconfigure to engage the idle SCPis in computational processes.

- **Assisting system goals**: Provide SCP
- **Actors**: SCPe, SCPis, applications
- **Targets**: SCPis involved in application executions
- **Scenario**: If a SCPi stays in idle mode for a specific period of time, it may request from other SCPis to take over some of the ongoing application executions.

For the *Provide Application* goal, are derived the following self-* objectives:

• **Self-Healing**: If a SCPi fails or is shut down, the applications executing on it must be made available on another SCPi in the SCPe hosting those SCPis.

- **Assisting system goals**: Provide Application
- **Actors**: SCPe, SCPis, applications
- **Targets**: applications transferred for execution to other SCPis
- **Scenario**: If a SCPi fails or is shut down while performing application executions, other SCPis shall detect the SCPi failure and shall take over the application executions carried by the failed SCPi.

• **Self-Configuring_2**: A SCPi is aware about the performance of the hosted applications. If an application is slowing down due to a lack of resources, this application can be distributed among different SCPis (run/resumed in parallel) if the application itself supports distributed execution.

- **Assisting system goals**: Provide Application
- **Actors**: SCPe, SCPis, application, resources
- **Targets**: application distributed for execution to other SCPis
- **Scenario**: If a SCPi detects low performance in application executions due to a lack of resources, the SCPi may request other SCPis to take over some of the hosted application executions, which will eventually release resources in the initial SCPi and improve the performance of its still hosted applications.

In addition to the self-* objectives derived from the context-specific GAR model, more self-* objectives might be derived from the constraints associated with the targeted system goal. Note that the analysis step in Figure 1 (see Section 2.3) uses the context-specific GAR model and elaborates on both system goal and constraints associated with that goal. Often environmental constraints introduce factors that may violate the system goals and self-* objectives will be required to overcome those constraints. Actually, such constraints might represent obstacles to the achievement of a goal. Constructing self-* objectives from goal constraints can be regarded as a form of *constraint programming*, in which a very abstract logic sentence describing a goal with its actors and targets (it may be written in a natural language as well) is extended to include concepts from *constraint satisfaction* and *system capabilities* that enable the achievement of the goal. In ARE, the capabilities are actually abstractions of system operations that need to be performed to maintain the goal fulfillment along with constraint satisfaction. In this approach, we need to query the provability of the targeted goal, which contains constraints, and then if the system goal cannot be fulfilled due to constraint satisfaction, a self-* objective is derived as an assistive system goal preserving both the original system's goal targets and constraint satisfaction.

An example demonstrating this process can be deriving self-* objectives from the Service Level

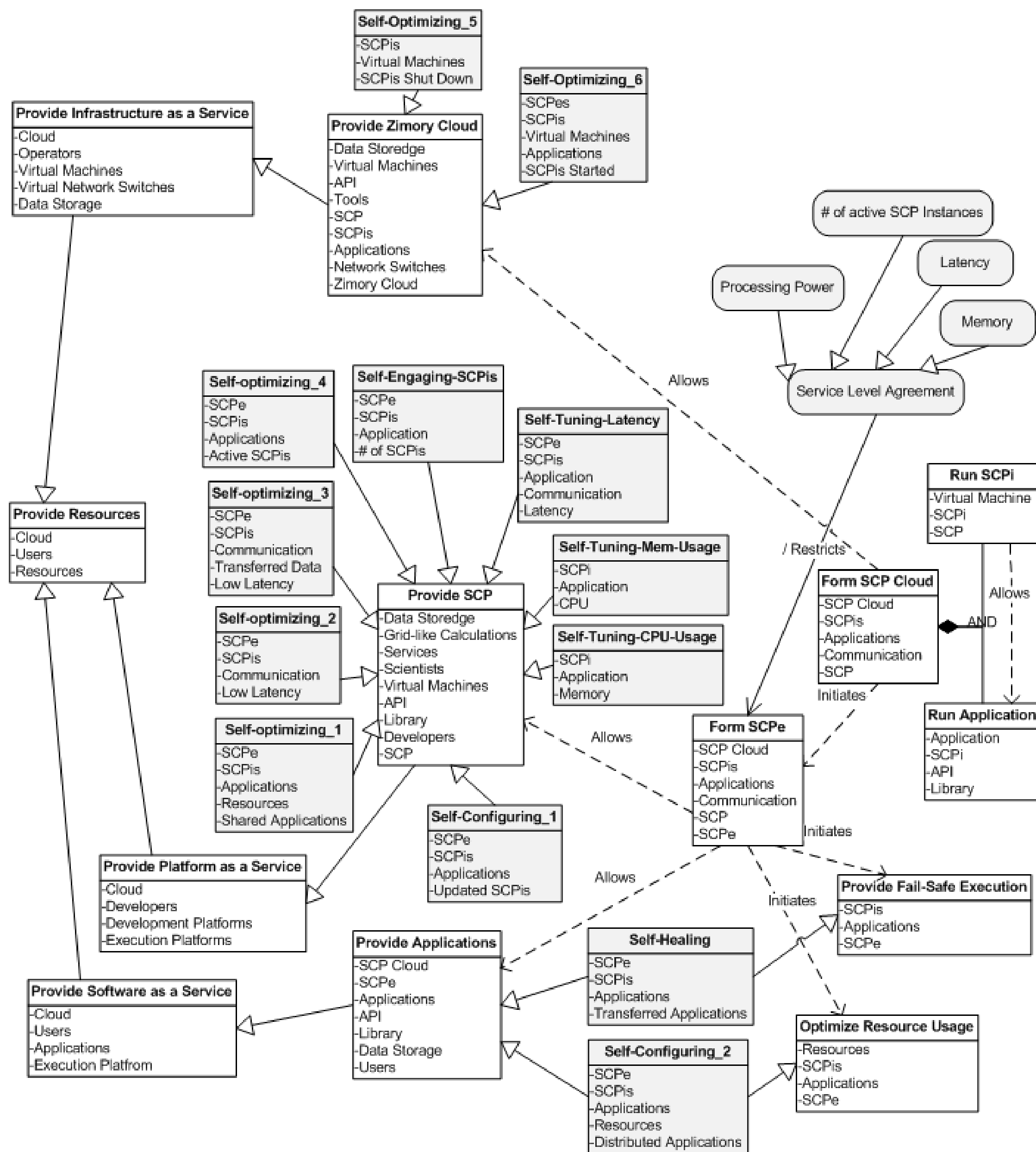

Figure 7. Science Clouds Goals Model with Self-* Objectives Assisting System Goals from Level 3

Agreement (SLA) constraints (see Section 4.2). SLA may impose constraints on application execution, e.g., certain number of active SCPis, certain latency between the communicating SCPis, or restrictions on processing power or on memory [22]. In this exercise are derived the following self-* objectives copying with the SLA constraints:

- **Self-Engaging-SCPis**: A SCPe formed for the execution of a certain application may need a certain number of involved SCPis.
  - **Assisting system goals**: Provide SCP
  - **Actors**: SCPe, SCPis, application
  - **Targets**: exact number of SCPis
  - **Scenario**: If an application requires an exact number of SCPis to run, then SCPe shall engage the exact number of SCPis needed for the execution of that application.
- **Self-Tuning-Latency**: A SCPe formed for the execution of a certain application may need a certain latency between the communicating SCPis needed for the execution of that application.
  - **Assisting system goals**: Provide SCP

  - **Actors**: SCPe, SCPis, application, communication
  - **Targets**: latency
  - **Scenario**: If an application requires a certain communication latency between the SCPis engaged to run that application, then each one of these SCPis shall maintain its communication latency by either speed up the communication (by applying the self-* objective Self-Optimizing_3) or slow it down (by introducing certain delay before sending the data packages).
- **Self-Tuning-CPU-Usage**: A SCPi executing a certain application might be restricted by maximum CPU power allowed to this application.
  - **Assisting system goals**: Provide SCP
  - **Actors**: SCPi, application
  - **Targets**: CPU power
  - **Scenario**: If an application is consuming more CPU power than the maximum allowed, then the hosting SCPi should slow down the application execution to minimize the CPU usage.
- **Self-Tuning-Memory-Usage**: A SCPi executing a certain application might be restricted by maximum memory allowed to this application.
  - **Assisting system goals**: Provide SCP
  - **Actors**: SCPi, application
  - **Targets**: memory
  - **Scenario**: If an application is consuming more memory than the maximum allowed, then the hosting SCPi should enforce lower memory use by this application.

Figure 7 depicts the Science Clouds Goals Model (shown in Figure 6), but enriched with the self-* objectives described above. As shown, these self-* objectives (depicted in gray color) inherit the system goals they assist by providing behavior alternatives with respect to these system goals. Note that, due to the "inheritance" relationship, the targets of the assisted system goals are kept in all of those self-* objectives. Note that the Science Clouds system switches to one of the assisting self-* objectives when alternative autonomous behavior is required (e.g., an SCPi fails to perform).

4.5. Formalizing Science Clouds with KnowLang

The next step after deriving the autonomy requirements per system goal is their specification with KnowLang. Note that the autonomy requirements carry all the necessary information that needs to be represented as knowledge for Science Clouds. Therefore, by specifying the captured self-* objectives, we build the necessary knowledge model for Science Clouds, which is the ultimate goal of this exercise. Specifying 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.

By applying the ARE approach to capture the autonomy requirements for Science Clouds, we actually perform the first two phases, as described above. This makes the resulting knowledge model very efficient and relevant and without any unnecessary knowledge details. KnowLang [4] is exclusively dedicated to knowledge specification where knowledge 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 for Science Clouds*, the first step is to specify the KB representing the cloud, SCPes, SCPis, applications, etc. To do that, we need to specify ontology structuring the knowledge domains of the cloud. 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) [4]. Note that being part of the autonomy requirements, knowledge plays a very important role in the expression of all the autonomy requirements: *autonomicity*, *knowledge*, *awareness*, *monitoring*, *adaptability*, *dynamicity*, *robustness*, *resilience*, and *mobility* outlined by GAR (see Section 2.3).

Science Cloud Ontology. Figure 8, depicts a graphical representation of the *Cloud_Thing* concept tree relating most of the concepts within the Science Cloud Ontology (SCCloud). Note that the relationships within a concept tree are "is-a" (inheritance), e.g., the *Latency* concept is a *Phenomenon* and the *Action* concept is a *Knowledge* and consecutively *Phenomenon*, etc. Most of the concepts presented in Figure 8 were derived from the Science Clouds Goals Model (see Figure 7). Other concepts are considered as "explicit" and were derived from

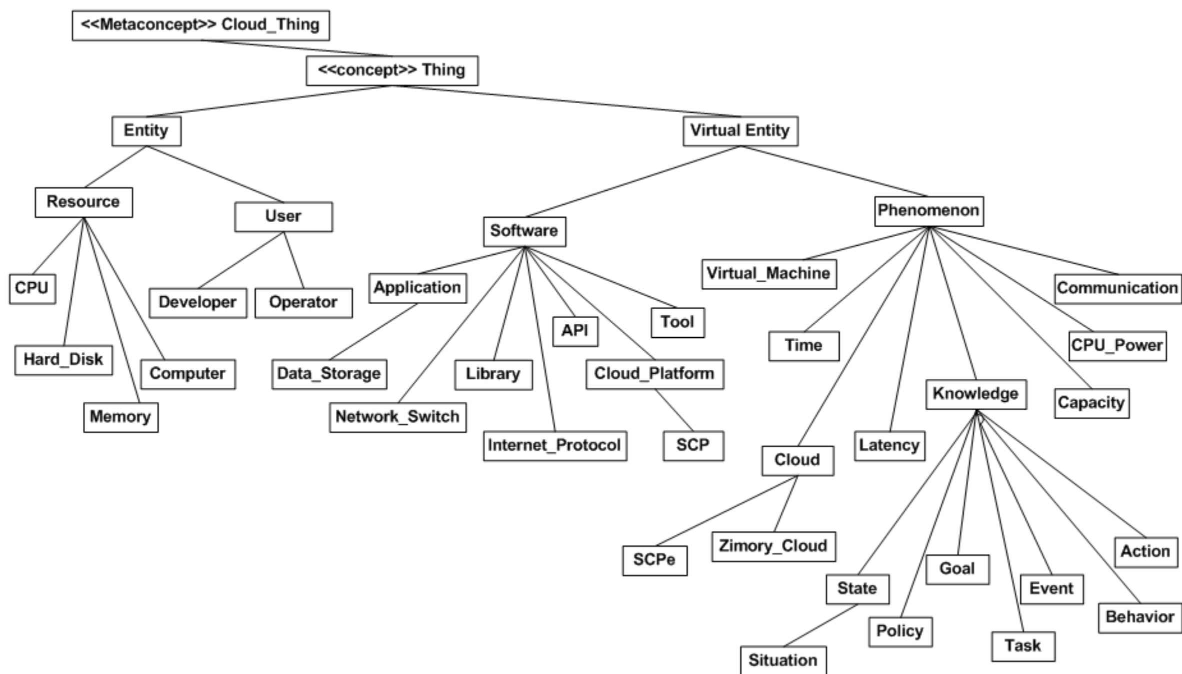

Figure 8. Science Clouds Ontology: Cloud_Thing Concept Tree

the KnowLang's multi-tier specification model [4]. The following is a sample of the KnowLang specification representing two important concepts: the SCP concept and the Application concept (partial specification only). 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 refers to the implementation of the concept in question, i.e., in the following example *SCPImpl* is the software implementation (presuming a C++ class) of the SCP concept.

```
// Science Cloud Platform
CONCEPT SCP {
  CHILDREN {}
    PARENTS { SCCloud.Thing..Cloud_Platform }
    STATES {
      STATE Running { this.PROPS.platform_API. STATES.Running AND
                      this.PROPS.platform_Library.STATES.Running }
      STATE Executing { IS_PERFORMING(this.FUNCS.runApp) }
      STATE Observing { IS_PERFORMING(this.FUNCS.runApp)  AND
                      SCCloud.Thing..Application.PROPS.initiator=this }
      STATE Down { NOT this.STATES.Running }
      STATE Overloaded { this.STATES.OverloadedCPU OR
                      this.STATES.OverloadedStorage OR this.STATES.OverloadedMemory }
      STATE OverloadedCPU { SCCloud.Thing..Metric.CPU_Usage.VALUE > 0.95 }
      STATE OverloadedMemory { SCCloud.Thing..Metric.Memory_Usage.VALUE > 0.95 }
      STATE OverloadedStorage { SCCloud.Thing..Metric.Hard_Disk_Usage.VALUE > 0.95 }
      STATE ApplicationTransferred { LAST_PERFORMED(this, this.FUNCS.transferApp) }
      STATE InCommunication { this.FUNCS.hasActiveCommunication }
      STATE InCommunicationLatency { this.STATES.InCommunication
                      AND this.FUNCS.getCommunicationLatency >0.5 }
      STATE InLowTrafic { this.FUNCS.getDataTrafic <= 0.5 }
      STATE Started { LAST_PERFORMED(this, this.FUNCS.start) }
      STATE Stopped { LAST_PERFORMED(this, this.FUNCS.stop) }
    }
    PROPS {
      PROP platform_API { TYPE {SCCloud.Thing..API} CARDINALITY {1} }
      PROP platform_Library { TYPE {SCCloud.Thing..Library} CARDINALITY {1} }
      PROP platform_CPU { TYPE {SCCloud.Thing..CPU} CARDINALITY {1} }
      PROP platform_Memory { TYPE {SCCloud.Thing..Memory} CARDINALITY {1} }
      PROP platform_Storage { TYPE {SCCloud.Thing..Data_Storage} CARDINALITY {1} }
      PROP platform_Applications { TYPE {SCCloud.Thing..Application} CARDINALITY {*} }
    }
    FUNCS {
      FUNC run { TYPE { SCCloud.Thing..Action.RunSCP } }
      FUNC down { TYPE { SCCloud.Thing..Action.StopSCP } }
      FUNC runApp { TYPE { SCCloud.Thing..Action.RunApplication } }
      FUNC startApp { TYPE { SCCloud.Thing..Action.StartApplication } }
      FUNC stopApp { TYPE { SCCloud.Thing..Action.StopApplication } }
      FUNC transferApp { TYPE { SCCloud.Thing..Action.TransferApplication } }
      FUNC startNewCommunication { TYPE { SCCloud.Thing..Action.StartCommunication } }
      FUNC stopNewCommunication { TYPE { SCCloud.Thing..Action.StopCommunication } }
      FUNC hasActiveCommunication { TYPE { SCCloud.Thing..Action.HasActiveCommunication } }
      FUNC getCommunicationLatency { TYPE { SCCloud.Thing..Action.GetCommunicationLatency } }
      FUNC getDataTraffic { TYPE { SCCloud.Thing..Action.GetTraffic } }
    }
    IMPL { SCCloud.SCPImpl }
  }

// Science Cloud Application
  CONCEPT Application {
    CHILDREN {}
          PARENTS { SCCloud.Thing..Software }
    STATES {
      STATE Running { PERFORMED(this.FUNCS.Started)
                      AND NOT PERFORMED(this.FUNCS. Stopped) }
      STATE Started { LAST_PERFORMED(this, this.FUNCS.start) }
      STATE Stopped { LAST_PERFORMED(this, this.FUNCS.stop) }
    }
    PROPS {
      PROP needed_CPU_Power { TYPE {SCCloud.Thing..CPU_Power} CARDINALITY {1} }
      PROP needed_Memory { TYPE {SCCloud.Thing..Capacity} CARDINALITY {1} }
      PROP needed_Storage { TYPE {SCCloud.Thing..Storage} CARDINALITY {1} }
      PROP distributiveness { TYPE {Boolean} CARDINALITY {1} }
      PROP requiredSCPis { TYPE {Integer} CARDINALITY {1} }
      PROP requiredLatency { TYPE { SCCloud.Thing..Latency } CARDINALITY {1} }
      PROP initiator { TYPE {SCCloud.Thing..SCP} CARDINALITY {1} }
    }
    FUNCS { ÅĚ         }
    IMPL { SCCloud.ApplicationImpl }
  }
```

As mentioned, the states are specified as Boolean expressions. For example, the state Executing is true while the SCP is performing the runApp 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., a SCP instance (SCPi). A complex state might be expressed as a Boolean function of other states. For example, the *Running* state is expressed as a

Boolean function of two other states, particularly, states of concept's properties, e.g., the SCP is running if both its API and Library are running:

```
STATE Running { this.PROPS.platform\_API.STATES.Running AND
                this.PROPS.platform\_Library.STATES.Running }
```

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. Note that to specify some of the SCP states, we used metrics. Metrics are special KnowLang constructs [4] that may handle the *monitoring autonomy requirements* (see Section 4.3).

```
STATE OverloadedCPU { SCCloud.Thing..Metric.CPU_Usage.VALUE > 0.95 }
```

The *Cloud_Thing* concept tree (see Figure 8) is the main concept tree of the SCCloud Ontology. Note that due to space limitations, Figure 8 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 the cloud, is the root of the concept tree shown in Figure 9. As shown, actions are grouped by subsystem (or part) they are associated with. For example, the SCP actions are: *RunSCP*, *StopSCP*, *LeaveSCPe*, and *JoinSCPe*.

Note that in the KnowLang specification models, in addition to concepts, we also specify *concept instances*, which are considered as objects and are structured in *object trees*. The latter are a conceptualization of how objects existing in the world of interest (e.g., Science Clouds) 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 [4]. Therefore, the object trees are the realization of concepts in the ontology domain (e.g., Science Clouds). 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. For example, the SCP concept might be regarded as a class and the SCPis as SCP "instances" of that class. In this exercise are specified a few exemplar SCPis as object trees, which are not presented here due to space limitations.

Autonomicity. To specify the self-*objectives (autonomicity requirements), we use goals, policies, and situations [4]. These are defined as explicit concepts in KnowLang and for the Cloud Ontology (SCCloud) are specified under the concepts *Virtual_entity->Phenomenon->Knowledge* (see Figure 8). Figure 10, depicts a concept tree representing the specified Science Clouds goals. Note that most of these goals were directly interpolated from the goals models (see Section 4.2) and more specifically, from the goals model for self-* objectives assisting the Science Clouds goals from Level 3 (see Section 4.4).

KnowLang specifies goals as *functions of states* where any combination of states can be involved. A goal has an *arriving state* (Boolean function of states) and an optional *departing state* (another Boolean function of states) [4]. 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 two simple goals. Note that their arriving and departing states can be either single SCP states or Boolean functions involving more than one state. Note that the states used to specify these goals are specified as part of the *SCP* concept.

```
//
//==== Cloud Goals ===========================================================
//
CONCEPT_GOAL Self-optimizing_1 {
  SPEC {
    DEPART { SCP.STATES.OverloadedCPU  }
    ARRIVE { SCP.STATES.ApplicationTransferred AND NOT SCP.STATES.OverloadedCPU }
  }
}
CONCEPT_GOAL Self-optimizing_3 {
  SPEC {
    DEPART { SCP.STATES.InCommunicationLatency }
    ARRIVE { SCP.STATES.InLowTrafic AND NOT SCP.STATES.InCommunicationLatency }
       }
}
```

According to the KnowLang semantics (see Section 3.1), in order to achieve specified goals (objectives), we need to specify *policies* triggering *actions* that will eventualy change the system states, so the desired ones, required by the goals, will become effective [4]. All the policies in KnowLang descend from the explicit *Policy* concept. Note that policies allow the specification of autonomic behavior (autonomic behavior can be associated with self-* objectives). 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 *ReduceCPUOverhead* - as the name says, this policy is intended to reduce the CPU overhead of a SCPi. As shown, the policy is specified to handle the goal *Self-Opimizing_1* and is triggered by the situation *HighCPUUsage*. Further, the policy triggers conditionally (the *CONDITONS* directive requires that a SCPi is executing an application) the execution of a sequence of actions.

```
CONCEPT_POLICY ReduceCPUOverhead {
  SPEC {
    POLICY_GOAL { SCCloud.Thing..Self-Optimizing_1 }
    POLICY_SITUATIONS { SCCloud.Thing..HighCPUUsage }
    POLICY_RELATIONS { SCCloud.Thing..Policy_Situation_1 }
    POLICY_ACTIONS {SCCloud.Thing..Action.StartCommunication,
                          SCCloud.Thing..Action.TransferApplication,
                    SCCloud.Thing..Action.StopCommunication }
    POLICY_MAPPINGS {
      MAPPING {
        CONDITIONS { SCCloud.Thing..SCP.STATES.Executing }
```

Figure 9. Science Clouds Ontology: Cloud_Action Concept Tree

```
        DO_ACTIONS { SCCloud.Thing..SCP.Action.StartCommunication,
                                                   SCCloud.Thing..SCP.Action.TransferApplication,
                     SCCloud.Thing..SCP.Action.StopCommunication }
      }
    }
  }
}
```

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. These situations shall be eventually outlined by the scenarios of the ARE Requirements Chunks (see Section 4.4). 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 or execution paths from that situation. The following code represents the specification of the *HighCPUUsage* situation, used for the specification of the *ReduceCPUOverhead* policy.

```
//
//==== Cloud Situations =========================================================
//
CONCEPT_SITUATION HighCPUUsage {
  CHILDREN {}
  PARENTS { SCCloud.Thing..Situation}
  SPEC {
    SITUATION_STATES { SCCloud.Thing..SCP.STATES.OverloadedCPU}
    SITUATION_ACTIONS { SCCloud.Thing..Action.TransferApplication,
                              SCCloud.Thing..Action.SlowDownApplication,
                        SCCloud.Thing..Action. StopApplication }
  }
}
```

As shown, the situation is 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 *HighCPUUsage* is effective if the SCP state *OverloadedCPU* is effective. The possible actions define what actions can be undertaken once the system falls in a particular situation. For example, the *HighCPUUsage* situation has three possible actions: *TransferApplication*, *SlowDownApplication*, and *StopApplication*. The following code represents another policy intended to handle the *HighCPUUsage* situation. In this policy are specified three *MAPPING* sections, which introduce three possible alternative execution paths.

```
CONCEPT_POLICY AIReduceCPUOverhead {
  SPEC {
    POLICY_GOAL { SCCloud.Thing..Self-Optimizing_1 }
    POLICY_SITUATIONS { SCCloud.Thing..HighCPUUsage }
    POLICY_RELATIONS { SCCloud.Thing..Policy_Situation_2 }
    POLICY_ACTIONS { SCCloud.Thing..Action.SlowDownApplication,
                               SCCloud.Thing..Action. StopApplication }
    POLICY_MAPPINGS {
      MAPPING {
        CONDITIONS { SCCloud.Thing..SCP.STATES.Executing }
        DO_ACTIONS { SCCloud.Thing..Action. SlowDownApplication }
        PROBABILITY {0.5}
      }
      MAPPING {
        CONDITIONS { SCCloud.Thing..SCP.STATES.Executing }
        DO_ACTIONS { SCCloud.Thing..Action. StopApplication }
        PROBABILITY {0.4}
      }
      MAPPING {
        CONDITIONS { SCCloud.Thing..SCP.STATES.Executing }
        DO_ACTIONS { GENERATE_NEXT_ACTIONS(SCCloud.Thing..SCP) }
        PROBABILITY {0.1}
      }
    }
  }
}
```

Note that situations are related to policies via relations [4]. The following code demonstrates how we related

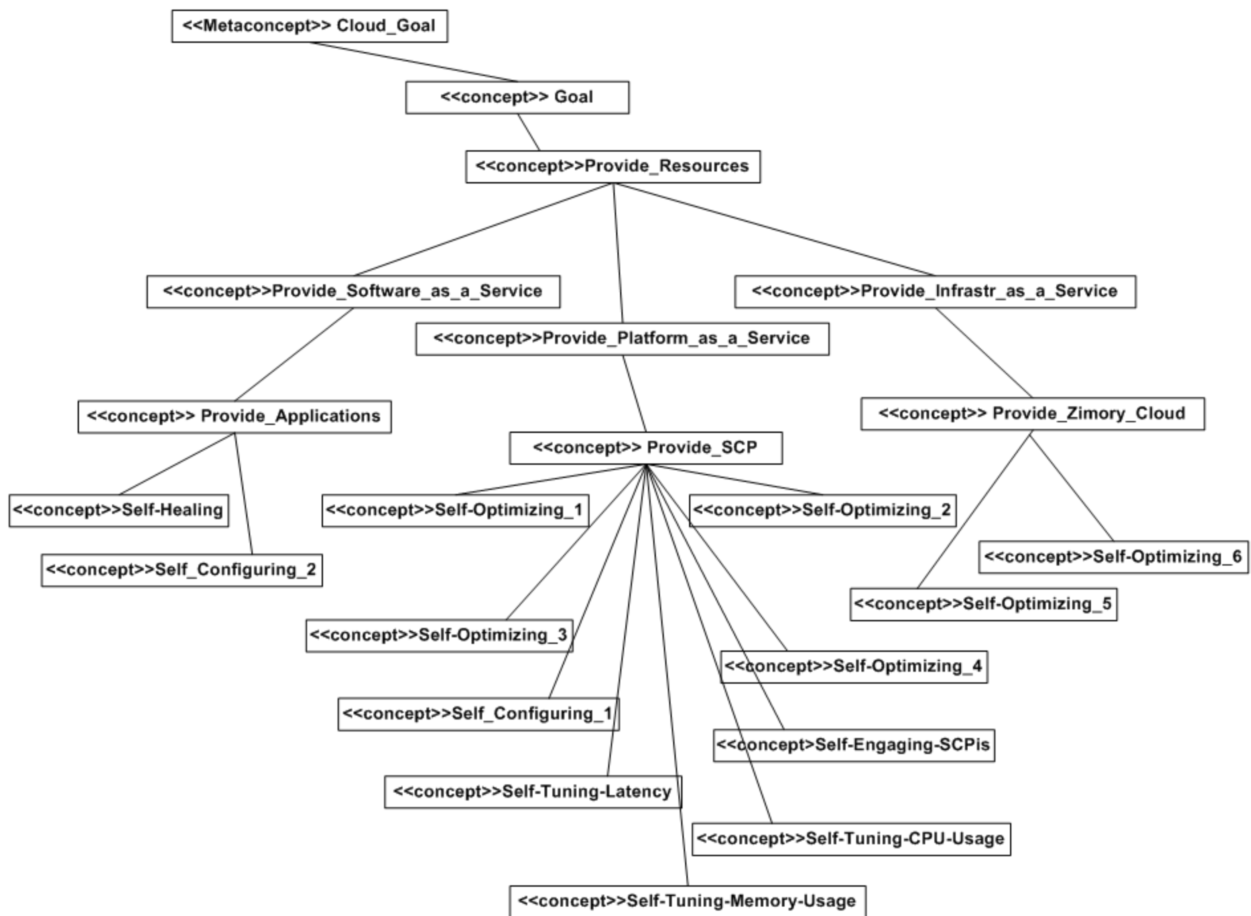

Figure 10. Science Cloud Ontology: Cloud_Goal Concept Tree

the *HighCPUUsage* situation to two different policies: *ReduceCPUOverhead* and *AIReduceCPUOverhead*.

```
//
//==== Cloud Relations =======================================================
//
RELATIONS {
  RELATION Policy_Situation_1 {
    RELATION_PAIR { SCCloud.Thing..HighCPUUsage, SCCloud.Thing..ReduceCPUOverhead }
                PROBABILITY {0.5}
  }
  RELATION Policy_Situation_2 {
    RELATION_PAIR { SCCloud.Thing..HighCPUUsage, SCCloud.Thing..AIReduceCPUOverhead}
                PROBABILITY {0.4}
  }
}
```

As specified, the probability distribution gives initial designer's preference about what policy should be applied if the system ends up in the *HighCPUUsage* situation. Note that at runtime, the KnowLang Reasoner maintains a record of all the action executions and re-computes the probability rates every time when a policy has been applied. Thus, although initially the system will apply the *ReduceCPUOverhead* policy (it has the higher probability rate of 0.5), if that policy cannot achieve its goal due to action fails (e.g., the communication link with another SCPi is broken and application transfer is not possible), then the probability distribution will be shifted in favor of the *AIReduceCPUOverhead* policy and the system will try to apply that policy. Note that in this case both policies share the same goal.

Probability distribution at the level of situation-policy relation can be omitted, presuming the relationship will not change over time. It is also possible to assign probability distribution within a policy where the probability values are set at the level of action execution, e.g., see the specification of the *AIReduceCPUOverhead* policy above. As specified, the *AIReduceCPUOverhead* policy is intended to handle the *HighCPUUsage* situation by providing alternative execution paths with similar probability distribution. Here, probabilities are recomputed after every action execution, and thus the behavior change accordingly. Moreover, to increase the goal-oriented autonomicity, in this policy's specification is used the special KnowLang operator *GENERATE_NEXT_ACTIONS*, which will automatically generate the most appropriate actions to be undertaken by the SCP. 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 [23] for probability assessment after action executions [4].

Note that when generating actions, the *GENERATE_NEXT_ACTIONS* operator follows a sequential decision-making algorithm where actions are selected to maximize the total reward. This means that the immediate reward of the execution of the first action, of the generated list of actions, might not be the highest one, but the overall reward of executing all the generated actions will be the highest possible one. Moreover, note that, the generated actions are selected from the predefined set of actions (e.g., the possible Cloud actions - see Figure 9). The principle of the decision-making algorithm used to select actions is as follows:

1. The average cumulative reward of the reinforcement learning system is calculated.
2. For each policy-action mapping, the KnowLang Reasoner learns the value function, which is relative to the sum of average reward.
3. According to the value function and *Bellman optimality principle*[1], is generated the optimal sequence of actions.

Monitoring. The *monitoring autonomy requirement* is handled via the explicit *Metric concept* [4]. 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 this approach is assumed that cloud 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 objects, instances of that class, we represent the real sensor. KnowLang allows the specification of four different types of metrics [4]:

- *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 both of *RESOURCE* and *QUALITY* type.

The following is a specification of metrics mainly used to assist the specification of states in the specification of the SCP concept (see Section 4.5).

```
//Cloud Metrics
CONCEPT_METRIC CPU_Usage {
  SPEC {        METRIC_TYPE { RESOURCE } METRIC_SOURCE { CPU.Usage }
    DATA { DATA_TYPE { Number } VALUE { 0.00 } }
  } }
CONCEPT_METRIC Memory_Usage {
  SPEC {        METRIC_TYPE { RESOURCE } METRIC_SOURCE { Memory.Usage }
    DATA { DATA_TYPE { Number } VALUE { 0.00 } }
  } }
CONCEPT_METRIC Hard_Disk_Usage {
  SPEC {        METRIC_TYPE { RESOURCE } METRIC_SOURCE { HDD.Usage }
    DATA { DATA_TYPE { Number } VALUE { 0.00 } }
  } }
```

5. Related work

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. 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 [24], 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. 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 runtime, 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 runtime) 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. The

[1] The Bellman optimality principle: If a given state-action sequence is optimal, and we were to remove the first state and action, the remaining sequence is also optimal (with the second state of the original sequence now acting as initial state).

Autonomic System Specification Language (ASSL) [25–27] is a framework providing for a formal approach to specifying and modeling autonomous (autonomic) systems by emphasizing the self-* requirements. Cheng and Atlee [28] report on work on specifying and verifying adaptive software. In [29, 30], research on runtime monitoring of requirements conformance is described. In [31], Sutcliffe, S. Fickas and M. Sohlberg demonstrate a method (called PC-RE) for personal and context requirements engineering that can be applied to autonomous systems. In addition, some research approaches have successfully used goal models as a foundation for specifying the autonomic behaviour [32] and requirements of adaptive systems [33].

A major breakthrough of the past decade in Software Requirements Engineering is the *goal-oriented approach* to capturing and analyzing stakeholder intentions to derive functional and non-functional (hereafter quality) requirements [34, 35]. In essence, this approach has extended upstream the software development process by adding a new phase (*early requirements analysis*) that is also supported by engineering concepts, tools and techniques.

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) [3] approach provides for analyzing 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. It also makes it more rational in that the designer can point to an explicit evaluation of these alternatives in terms of stakeholder criteria to justify her choice.

ARE uses GORE as the first phase of the Autonomy Requirements Engineering process. ARE uses GORE to build goal models that can help us derive autonomy requirements in several ways:

1. Goal models can be used to capture and refine requirements for autonomic systems. A goal model provides the starting point for the development of such a system 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. Thus, requirements goal models can be used as a baseline for validating software systems.

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 stakeholder quality concerns and other constraints. This allows for exploration and analysis of alternative system behaviors at design time, which leads to more predictable and trusted autonomic systems. It also means that if the alternatives that are initially delivered with the system perform well, there is no need for complex social interactions among autonomic elements. 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. Thus, 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 minimize the need for that difficult task.

3. Goal models provide the traceability mechanism from AC system designs to stakeholder requirements. When a change in stakeholder 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. For instance, if a change in stakeholder requirements affected a particular goal in the model, it is possible to see how this goal is decomposed and which components/autonomic elements implementing the goal are in turn affected. By analyzing the goal model, it is also easy to identify how a failure to achieve some particular goal affects the overall objective of the system. At the same time, 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 autonomic elements to high-level system objectives and quality concerns. These high-level objectives or quality concerns serve as the common knowledge shared among the autonomic computing elements 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.

6. Conclusions

This article has presented an Autonomy Requirements Engineering approach, developed to tackle the special

autonomy requirements used to derive self-adaptation capabilities of software intensive systems. A proof-of-concept example has been presented where is applied the proposed ARE model to a Science Clouds case study. With this example is demonstrated how ARE can be used to both elicit and express autonomy requirements for software-intensive, yet self-adaptive, systems. Note that ARE relies on Goal-Oriented Requirements Engineering (GORE) to elicit and define the system goals, and uses a Generic Autonomy Requirements (GAR) model to derive and define *assistive* and eventually *alternative goals* (or objectives) of the system. The system may pursue these "*self-* objectives*" in the presence of factors threatening the achievement of the initial system goals. Once identified, the autonomy requirements, including the self-* objectives, have been further specified with KnowLang. The specification of the Science Clouds autonomy requirements along with accompanying rationale have also been presented in this article.

Future work is mainly concerned with development of tools for ARE. An efficient ARE Tool Suite incorporating an *autonomy requirements validation* approach is the next logical step needed to complete the ARE Framework. Moreover, an efficient ARE Framework shall adopt KnowLang as a formal notation and provide tools for specification and validation of autonomy requirements. Runtime knowledge representation and reasoning shall be provided along with monitoring mechanisms to support the autonomy behavior of a system at runtime. We need to build an ARE Test Bed tool that will integrate the KnowLang Reasoner and will allow for validation of self-* objectives based on simulation and testing. This will help engineers validate self-* objectives by evaluating the system's ability to perceive the internal and external environment and react to changes. Therefore, with the ARE Test Bed tool, we shall be able to evaluate capabilities that might manifest system awareness about situations and conditions. Ideally, both the autonomy requirements model specified in the form of knowledge representation and the reasoner, can be further implemented in autonomous spacecraft as an engine responsible for the adaptive behavior. Eventually, a code generator shall be able to generate stubs supporting the operations of the KnowLang Reasoner. These stubs can be further used as a basis for the real implementation of the mechanism controlling the autonomic behavior of the system.

Acknowledgement. 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 Center at University of Limerick, Ireland.

References

[1] Vassev, E. and Hinchey, M. (2014) *Autonomy Requirements Engineering for Space Missions*, NASA Monographs in Systems and Software Engineering (Springer). doi:10.1007/978-3-319-09816-6, URL http://dx.doi.org/10.1007/978-3-319-09816-6.

[2] Vassev, E. and Hinchey, M. (2013) Autonomy requirements engineering. *IEEE Computer* **46**(8): 82–84.

[3] van Lamsweerde, A. (2000) Requirements engineering in the Year 00: A research perspective. In *Proceedings of the 22nd IEEE International Conference on Software Engineering (ICSE-2000)* (ACM): 5–19.

[4] Vassev, E., Hinchey, M., Montanari, U., Bicocchi, N., Zambonelli, F. and Wirsing, M. (2012), D3.2: Second Report on WP3: The KnowLang Framework for Knowledge Modeling for SCE Systems. ASCENS Deliverable.

[5] Vassev, E. and Hinchey, M. (2013) On the autonomy requirements for space missions. In *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing Workshops (ISCORCW 2013)* (IEEE Computer Society).

[6] Vassev, E. and Hinchey, M. (2013) Autonomy requirements engineering: A case study on the BepiColombo Mission. In *Proceedings of the C* Conference on Computer Science & Software Engineering (C3S2E'13)* (ACM): 31–41.

[7] Vassev, E. and Hinchey, M. (2013) Autonomy requirements engineering. In *Proceedings of the 14th IEEE International Conference on Information Reuse and Integration (IRI'13)* (IEEE Computer Society): 175–184.

[8] ASCENS (2012), ASCENS - Autonomic Service-Component Ensembles. Url: http://www.ascens-ist.eu/.

[9] Vassev, E. and Hinchey, M. (2014) Modeling swarm robotics with KnowLang. In *Proceedings of the 2nd International Workshop on Formal Methods for Self-Adaptive Systems (FMSAS 2014)* (Springer).

[10] Vassev, E., Hoch, N., Bensler, H. and Hinchey, M. (2014) Formalizing eMobility with KnowLang. In *Proceedings of C* Conference on Computer Science and Software Engineering (C3S2E 2014)* (ACM): 27–34.

[11] Vassev, E., Mayer, P. and Hinchey, M. (2014) Formalizing self-adaptive clouds with KnowLang. In *Proceedings of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2014)* (Springer).

[12] Ross, D. and Schoman, K. (1977) Structured analysis for requirements definition. *IEEE Transactions on Software Engineering* **3**(1): 6–15.

[13] van Lamsweerde, A., Darimont, R. and Massonet, P. (1995) Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In *Proceedings of the 2nd International IEEE Symposium on Requirements Engineering* (IEEE): 194–203.

[14] Haumer, P., Pohl, K. and Weidenhaupt, K. (1998) Requirements elicitation and validation with real world scenes. *IEEE Transactions on Software Engineering - Special Issue on Scenario Management* : 1036–1054.

[15] Rolland, C., Souveyet, C. and Achour, C. (1998) Guiding goal-modeling using scenarios. *IEEE Transactions on Software Engineering - Special Issue on Scenario Management* : 1055–1071.
[16] van Lamsweerde, A. (2000) Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'2000)* (ACM): 5–19.
[17] Neapolitan, R. (2003) *Learning Bayesian Networks* (Prentice Hall).
[18] Vassev, E. and Hinchey, M. (2012) 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)* (IEEE Computer Society): 156–163.
[19] Vassev, E., Hinchey, M. and Gaudin, B. (2012) Knowledge representation for self-adaptive behavior. In *Proceedings of C* Conference on Computer Science & Software Engineering (C3S2E '12)* (ACM): 113–117.
[20] Vassev, E. (2012) *KnowLang Grammar in BNF*. Tech. Rep. Lero-TR-2012-04, Lero, University of Limerick, Ireland.
[21] Mayer, P., Klarl, A., Hennicker, R., Puviani, M., Tiezzi, F., Pugliese, R., Keznikl, J. *et al.* (2013) The autonomic cloud: A vision of voluntary, peer-2-peer cloud computing. In *Proceedings of the 3rd Workshop on Challenges for achieving Self-Awareness in Autonomic Systems* (Philadelphia, USA): 1–6.
[22] Serbedzija, N., Reiter, S., Ahrens, M., Velasco, J., Pinciroli, C., Hoch, N. and Werther, B. (2011), D7.1: First Report on WP7 Requirement Specification and Scenario Description of the ASCENS Case Studies. ASCENS Deliverable.
[23] Ewens, W. and Grant, G. (2005) Stochastic processes (i): Poison processes and Markov chains. In *Statistical Methods in Bioinformatics, 2nd edition, Springer, New York*.
[24] Vassev, E., Hinchey, M., Balasubramaniam, D. and Dobson, S. (2011) An ASSL approach to handling uncertainty in self-adaptive systems. In *Proceedings of the 34th annual IEEE Software Engineering Workshop (SEW 34)* (IEEE Computer Society): 11–18.
[25] Vassev, E. (2009) *ASSL: Autonomic System Specification Language - A Framework for Specification and Code Generation of Autonomic Systems* (LAP Lambert Academic Publishing, Germany).
[26] Vassev, E. (2008) *Towards a Framework for Specification and Code Generation of Autonomic Systems*. Ph.D. thesis, Computer Science and Software Engineering Department, Concordia University, Quebec, Canada.
[27] Vassev, E. and Hinchey, M. (2009) ASSL: A software engineering approach to autonomic computing. *IEEE Computer* **42**(6): 106–109.
[28] Cheng, B. and Atlee, J. (2007) Research directions in requirements engineering. In *Proceedings of the 2007 Conference on Future of Software Engineering (FOSE 2007)* (IEEE Computer Society): 285–303.
[29] Fickas, S. and Feather, M. (1995) Requirements monitoring in dynamic environments. In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE 1995)* (IEEE Computer Society): 140–147.
[30] Savor, T. and Seviora, R. (1997) An approach to automatic detection of software failures in real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium* (IEEE Computer Society): 136–147.
[31] Sutcliffe, A., Fickas, S. and Sohlberg, M. (2006) PC-RE a method for personal and context requirements engineering with some experience. *Requirements Engineering Journal* **11**: 1–17.
[32] Lapouchnian, A., Yu, Y., Liaskos, S. and Mylopoulos, J. (2006) Requirements-driven design of autonomic application software. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2006)* (ACM): 7.
[33] Goldsby, H., Sawyer, P., Bencomo, N., Hughes, D. and Cheng, B. (2008) 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).
[34] Dardenne, A., van Lamsweerde, A. and Fickas, S. (1993) Goal-directed requirements acquisitions. *Science of Computer Programming* **20**: 3–50.
[35] Mylopoulos, J., Chung, L. and Nixon, B. (1992) Representing and using non-functional requirements: a process-oriented approach. *IEEE Transactions on Software Engineering* **18**(6): 483–497.