www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D3.3: Third Report on WP3

Knowledge Modeling for ASCENS Case Studies and KnowLang Implementation

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 2.2 (30.7.2011)**

Lead contractor for deliverable: **UL**
Author(s): **Emil Vassev (UL), Mike Hinchey (UL), Nicola Bicocchi (UNIMORE), Philip Mayer (LMU)**

Reporting Period: **3**
Period covered: **October 1, 2012 to September 30, 2013**
Submission date: **November 8, 2013**
Revision: **Final**
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

SEVENTH FRAMEWORK PROGRAMME

Executive Summary

One of the main scientific contributions that we expect to achieve with ASCENS is related to Knowledge Representation and Reasoning for self-adaptive systems. Within the WP3's mandate of the project, we are currently developing the KnowLang Framework that implies a notion for modeling knowledge and self-adaptive behavior of ASCENS-like systems. In this third year of WP3, we continued working on the implementation of KnowLang where we focused on the KnowLang Toolset. Moreover, we started working on the implementation of the KnowLang Reasoner carrying an integrated mechanism for knowledge processing and self-adaptive behavior. In addition to this routine implementation, we also worked on the knowledge models for the ASCENS Case Studies. In a joint project with ESA (European Space Agency), we developed an approach to Autonomy Requirements Engineering (ARE), which we used to build efficient and relevant knowledge models for ASCENS. In this endeavor, we used ARE as a software engineering step to select and refine relevant and efficient knowledge data that needs to be represented with KnowLang for the ASCENS Science Clouds case study. The novel requirements engineering technique helped us build a knowledge representation model, which is at the right level of abstraction and relevance, i.e., carrying all the necessary details to represent the knowledge necessary to process self-adaptive behavior based on awareness capabilities. Note that, finding the right level of abstraction and data relevance for the knowledge models specified with KnowLang was a major flow in our previous knowledge models, which carried unnecessary details, thus eventually overwhelming the reasoning process. Similar to the previous years, in this year, we continued collaborating with WP1, WP2, WP4 and WP7 for the gradual integration of KnowLang with SCEL and SOTA tackled by WP1 and WP4 respectively, and for knowledge modeling for the ASCENS Case Studies tackled by WP7.

Contents

1 Introduction

Knowledge and awareness are two major and necessary compounds that can make a software-intensive system self-adaptive. WP3 focuses on approaches and mechanisms dealing with these issues. Three years ago, when we started this project, the major challenge was ”how to represent knowledge for self-adaptive systems”. In this third year of the project, we may say that we have found our answer to the ”how” question - we developed the KnowLang framework where symbols serve as knowledge surrogates for real world artefacts and underlying mechanisms provide vital connection between knowledge, perception and actions realizing self-adaptive behavior. However, somewhere along the route of our research, we realized that another major issue is the relevance of the knowledge to be presented. Thus, our focus slightly shifted from ”how” to ”what”, i.e., what knowledge should be given to the system in order to become self-adaptive. Along with our work on the implementation of KnowLang, this deliverable presents our approach to this new issue.

1.1 Research Focus and Tasks

In this third year of WP3, we continued working on the implementation of KnowLang (WP3.T1) where we focused on the KnowLang Toolset. We fully implemented the KnowLang's Text Editor, Grammar Compiler and Parser, and almost completed the implementation of the Visual Editor, Semantic Analyzer and Consistency Checker. Moreover, we started working on the implementation of the KnowLang Reasoner (WP3.T3) carrying the integrated mechanism for knowledge processing and self-adaptive behavior along with ASK and TELL operators and implementation of the awareness control loop. In addition to this routine implementation, we also worked on the knowledge models for the ASCENS Case Studies (WP3.T2). In a joint project with ESA (European Space Agency), we developed an approach to Autonomy Requirements Engineering (ARE), which we used to build efficient and relevant knowledge models for ASCENS. In this endeavor, we used ARE as a software engineering step to select and refine relevant and efficient knowledge data that needs to be represented with KnowLang for the ASCENS Science Clouds case study. The novel requirements engineering technique helped us build a knowledge representation model, which is at the right level of abstraction, i.e., carrying all the necessary details to represent the knowledge necessary to process self-adaptive behavior based on awareness capabilities. Note that, finding the right level of abstraction for the knowledge models specified with KnowLang was a major flow in our previous knowledge models, which carried unnecessary details, thus eventually overwhelming the reasoning process. With ARE, we focus on the so-called self-* objectives providing for self-adaptive behavior and consecutively centering the knowledge models around this self-adaptive behavior, which makes the knowledge representation very efficient.

1.2 Relations with Other WPs

In the third year of this project, we continued collaborating intensively with WP1, WP2, WP4 and WP7. The collaboration with WP1 is currently going at the level of interoperability between SCEL and KnowLang. KnowLang provides a KR model of the SCEL knowledge base and the Knowlang Reasoner should be properly integrated with SCEL. We are currently working on a solution where the KnowLang Reasoner communicates with a SCEL application via a distinct tuple space where data initiating ASK and TELL operations is recorded. Basically, the application writes in this tuple space errors, sensory data and actions performed by the system. Moreover, when needed, the SCEL program gets from this tuple space a self-adaptive behavior model outlined by a sequence of actions to be performed by the system. We are also working on the integration of POEM with the reasoning process driven by the KnowLang Reasoner. Our goal is to integrate POEM as a FOL reasoner. To do so, we are currently working on a special interpreter that extracts from a KnowLang specification the specified facts, rules and constraints to supply these to the POEM Reasoner, which can quantify on them under request.

The collaboration with WP2 continued with further implementation of our model for soft constraints for KnowLang [VHM$^+$12]. The soft constraints for KnowLang are used as a KR technique that will help designers impose constraining requirements for special liveness properties, an approximation to our understanding of good-to-have properties. The approach associates tuples of possible values held by special KnowLang variables with possible preferences.

Concerning WP4, in this third year of the project, we relied on WP4 mainly to determine scenarios for our self-* objectives by using SOTA, the State Of The Affairs framework tackled by WP4. Moreover, WP4 has compiled an extensive catalogue of adaptation patterns, which we also used to derive some self-adaptation scenarios for the Science Clouds case studies. SOTA adaption patterns are more abstract than ARE scenarios. Ideally, they describe how different components might interact (or might be connected) to achieve adaptive behavior. To do so, SOTA provides special *trajectories* towards a goal [PNAZ13]. These trajectories directly connect SOTA and ARE, because we may translate trajectories to ARE scenarios where given a set of predefined conditions (an n-dimensional volume in SOTA), ARE scenarios are executed causing the system to move within a SOTA space. Different scenarios imply different trajectories.

WP7 [SMP$^+$12, SHP$^+$13] provides vital experimental platforms for both the notation and toolset of KnowLang. In collaboration with WP7, this year, we used ARE to capture relevant knowledge data and we used KnowLang to specify a complete relevant knowledge model for the Science Clouds case study where WP7 provided us with important information related to the possible states expression. Note that with the ARE approach we overcome the main challenge to identify the right level of knowledge abstraction and relevance at which reasoning can provide for adaptation and self-awareness. For the Science Clouds case study, along with the intensive specification of initial knowledge models (ontologies, facts, rules and constraints), we also specified complete behavior models provided by policies and situations. These models actually specify self-* objectives derived with the ARE approach (see Section 3). In the following last year of the project, we will build relevant knowledge models for the other two case studies. These models along with the KnowLang Reasoner will be used for awareness prototyping, which is planned for Task 3.4. Note that currently we do have previously specified knowledge models for these two case studies, but we still need to complete and refine these by applying the ARE approach.

WP8 tackles the Ensemble Development Life Cycle (EDLC) [HK13, KHK$^+$13]. The ARE approach contributes to EDLC by adding on to the requirements engineering of the design phase of EDLC. More specifically, it helps to capture the autonomy requirements of the system in question, which in turn are used as a basis for deriving relevant knowledge-representation models for that very system. In the EDLC requirements engineering, both SOTA and ARE are collaborating to come up with self-adaptive behavior. ARE's GAR model might be used to add on to the SOTA adaptation patterns by outlining self-* objectives providing for self-adaptive behavior. Moreover, SOTA patterns might help identify self-* objectives along with proper scenarios defined as SOTA trajectories to a goal.

1.3 Structure of the Document

The rest of this document is organized as follows. Section 2 presents the Autonomy Requirements Engineering approach by demonstrating how autonomy requirements can be captured by focusing on self-* objectives. Section 3 presents our work on capturing the autonomy requirements for the Science Clouds case study with ARE and specifying these requirements with KnowLang as a relevant knowledge representation model. In Section 4, we present our results related to the implementation of the KnowLang Framework, those including both the KnowLang Toolset and Reasoner. Finally, to conclude the topic, in Section 5, we present a brief summary and future goals.

2 Autonomy Requirements Engineering

The Autonomy Requirements Engineering (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 *generic autonomy requirements* (GAR) with *goal-oriented requirements engineering* (GORE). Using this approach, software engineers can determine what autonomic features to develop for a particular system (e.g., the ASCENS Science Clouds case study) as well as what artifacts that process might generate (e.g., goals models, requirements specification, etc.). For the ASCENS project in particular, ARE shall not only help us capture the autonomy requirements for the ASCENS Case Studies, but also help us derive efficient and relevant knowledge models for these case studies.

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 [VH13d, VH13c, VH13a, VH13b]: 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 level 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 our 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 [RS77]. 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 [vLDM95, HPW98, RSA98]. 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 our experience, goals can also be identified systematically by searching for intentional keywords in the preliminary documents provided, e.g., ASCENS case study description. 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 [vL00]. 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 [VH13d, VH13c, VH13a, VH13b]:

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* [VH13b] 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 [VH13c, VH13b]. 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. For example, the context-specific GAR model for the domain of Cloud Computing defines a predefined set of self-* objectives that cope with both constraints and challenges a cloud must overcome while delivering resources to its users. For example, GAR may define the following self-* objectives for Scientific Clouds (see Section 3.2):

- *self-healing*: The high-level goal of the science cloud is to provide application execution. If a cloud machine fails or is shut down, the applications executing must be made available (resumed) somewhere else in the cloud. A self-healing self-* objectives should ensure that this is possible.

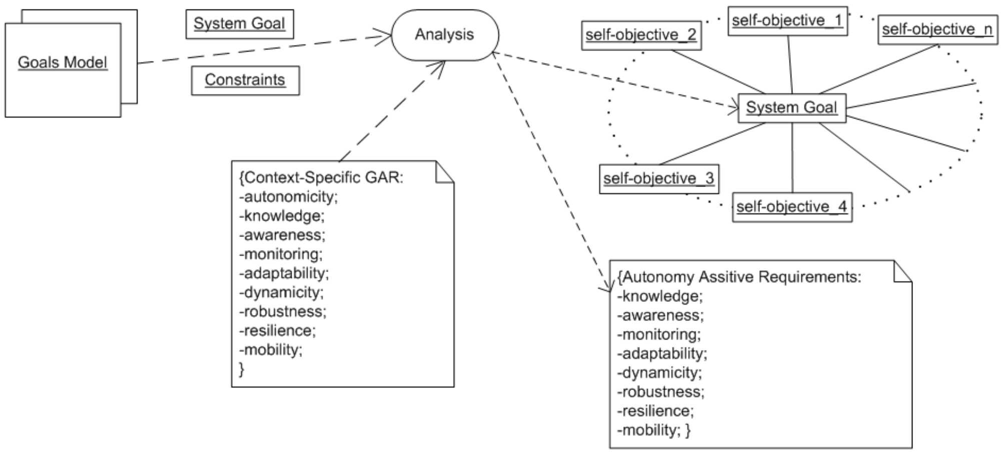


Figure 1: The ARE Process of Deriving Self-* Objectives per System Goal

- *self-configuring*: Each cloud machine is aware about changes in the cloud - new machines can be added to the cloud or other can be opted out. A cloud machine should adapt itself to make use of newly available resources or consider disappearing resources.

- *self-optimizing*: If a cloud machine reaches its capacity (consistent high CPU load or swapping), it may transfer some of the load to another cloud machine. The same applies to overloaded links in the network.

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 [VH13d, VH13b] 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 events and/or sensor signals to derive particular system states.

- *Resilience* and *robustness* - a sort of soft-goals. For example, such requirements for Science Clouds can be defined as "*robustness: cloud is robust to communication latency*" and "resilience: cloud is resilient to cloud machines failures, disappearances, or appearances". These requirements can be specified as soft goals leading the system towards "*reducing and copying with communication latency*" and "*keeping cloud's performance optimal*". A soft goal is satisficed 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 satisficed to a sufficient degree. Thus, when specifying robustness and resilience autonomy requirements we need to set the desired degree of satisfaction, e.g., by using probabilities.

- *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 [VH13c]:

- **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 "textitautonomy 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.*

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*.

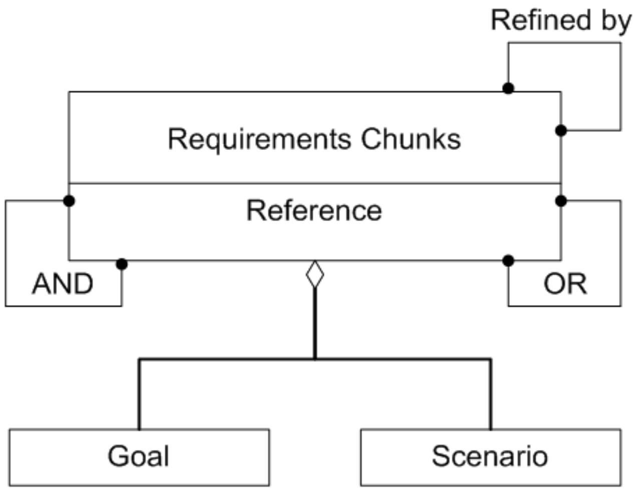


Figure 2: Requirements Chunk - Goal & Scenario

2.5 Formal Specification

ARE relies on KnowLang for the formal specification of the elicited autonomy requirements. Therefore, 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 Capturing Autonomy Requirements for Science Clouds

In this exercise, we applied the ARE approach to capture the autonomy requirements for the ASCENS Science Clouds case study. This helped us determine the right level of abstraction for the needed knowledge representation model. By applying GORE, we built *goals models* that helped us consecutively derive and organize the *autonomy requirements* for Science Clouds. In our approach, the goals models provided the starting point for ARE Science Clouds by defining 1) the objectives of the system in 2) the system's operational environment, and by identifying the 3) restrictions that exist in this environment along with service-level agreements as well as 4) the immediate targets supporting the system objectives and 5) constraints the system needs to address. Moreover, GORE helped us identify the *system actors* (SCP, SCP instance, virtual machine, SCP ensemble, etc.). In this exercise, we did not categorize the objectives' actors, but for more comprehensive requirements engineering, actors might be categorized by role or by importance (e.g., main, supporting and offstage actors). Note that the ARE goals models can be used as a baseline for validating the system.

Next, following the ARE Approach, we put the GAR model in the context of *cloud computing* to derive a domain-specific GAR model. Further, we merged the domain-specific GAR model with the goals model to derive for the system goals assistive and alternative self-* objectives along with appropriate adaptation-assistive attributes. Next, we identified the autonomy needs and the ARE requirements chunks for Science Clouds. Finally, we specified those requirements chunks with KnowLang, which basically means that we built the knowledge representation model for Science Clouds at the right level of abstraction.

Both ASCENS deliverables D7.2 [SMP+12] and D7.3 [SHP+13] were the major source of information for this activity. The description of the Science Clouds case study along with that of the *cloud computing* domain helped us build our GAR model for Science Clouds (see Section 3.2). Moreover, D7.2 was used as a preliminary source for determining the goals (or objectives) of a Science Cloud (see Section 3.1). In particular, we derived most of the objectives from the services a Science Cloud needs to provide. Further, self-* objectives assisting the cloud's objectives were derived from the prospective self-adaptive and autonomy cloud's features described in D7.2. Finally, the scenario described in D7.3 inspired some of the scenarios we defined for the ARE requirements chunks for Science Clouds. Note that these scenarios along with the science cloud goals and self-* objectives were formally specified with KnowLang (see Section 3.4).

3.1 GORE for Science Clouds

Science Clouds is a cloud computing scientific platform for application execution and data storage [MKH+13]. 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 so-called 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 [SRA+11]. 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 [SRA+11]. 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 deduct that the Science Clouds' main objective is to *provide a scientific platform for application execution and data storage* [MKH+13]. 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.

The three different services above can be defined as three main goals of cloud computing, and their realization by Science Clouds will define the main Science Clouds goals. Figure 3 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 deduct 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 3, 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 3 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

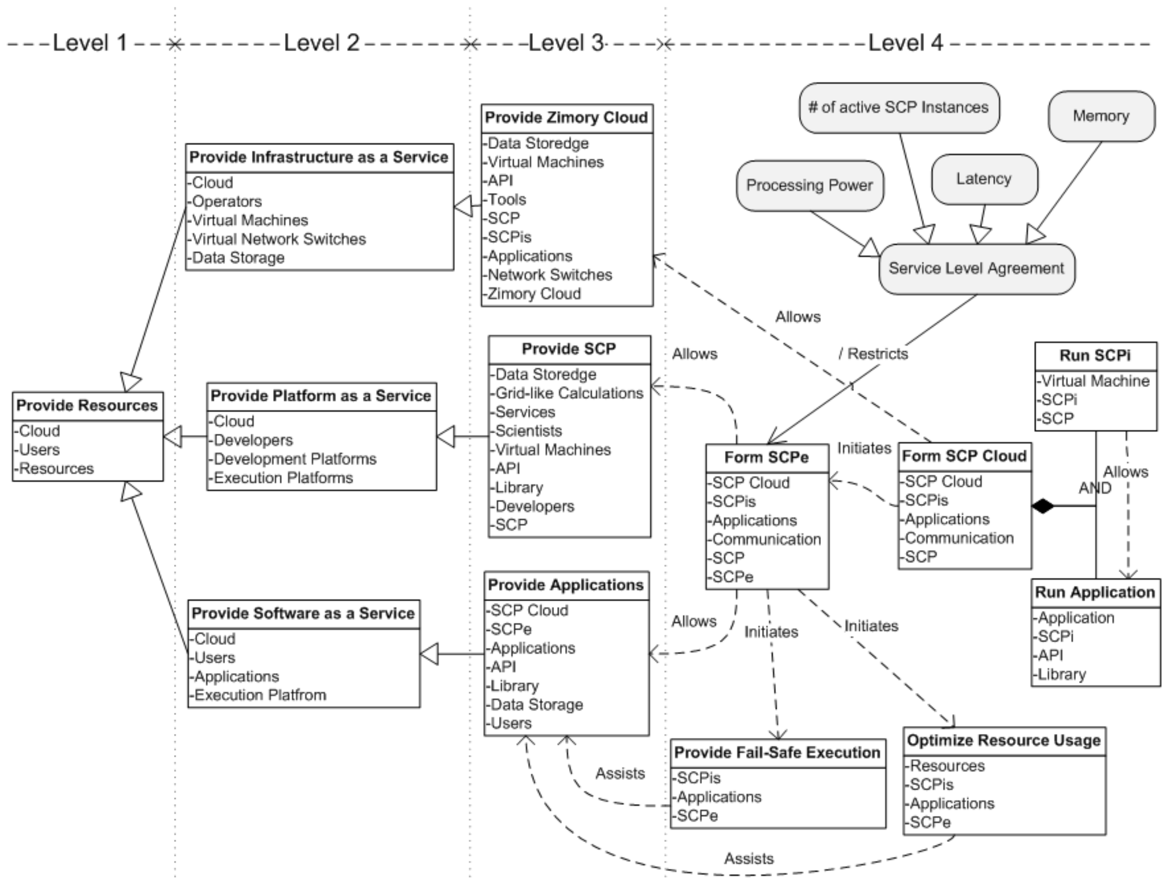


Figure 3: Science Clouds Goals Model

describe the system goals by goal levels as shown in Figure 3:

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. 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 3). 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 3). 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 3).
  - **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

3.2 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 elaborated 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 [MKH$^+$13]:

- *fail-safe operation*: An application should be available even its host SCPi fails (see Provide Fail-safe Execution goal in Section 3.1).
- *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 3.1) will result into self-adaptive behavior realized by self-* objectives. These self-* objectives along with the autonomy-assistive requirements form our 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);

3.3 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 3 and Section 3.1) present the main system goals, we applied 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 3.2 are relevant to every one of these three goals. In this section, we present the self-* objectives derived for these three goals. The self-* objectives are presented as autonomy requirements chunks (see Section 2.4).

For the *Provide Zimory Cloud* goal we 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 we 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 we 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 Agreement (SLA) constraints (see Section 3.1). SLA may impose constraints on application execution, e.g., certain number of active SCPis, certain latency between the communicating SCPis, or

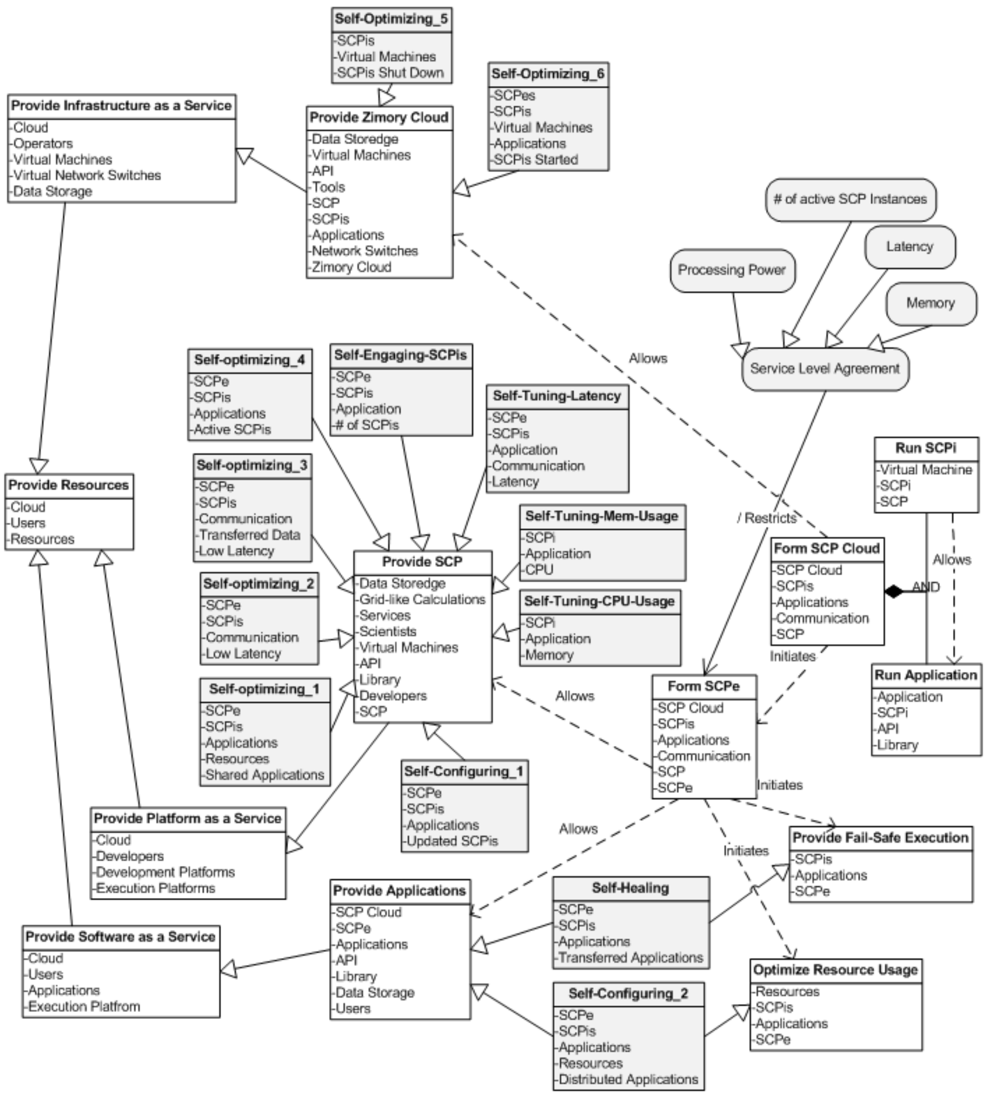


Figure 4: Science Clouds Goals Model with Self-* Objectives Assisting System Goals from Level 3

restrictions on processing power or on memory [SRA+11]. In this exercise, we 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 4 depicts the Science Clouds Goals Model (shown in Figure 3), 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).

3.4 Knowledge Representation for 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. Recall that specifying with KnowLang goes over a few phases [12]:

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, which was a major problem in our previous knowledge models for the ASCENS case studies. Recall that KnowLang [VHM+12] 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) [VHM+12]. 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).

3.4.1 Science Cloud Ontology

Figure 5, 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 5 were derived from the Science Clouds Goals Model (see Figure 4). Other concepts are considered as "explicit" and were derived from the KnowLang's multi-tier specification model [VHM+12]. The following

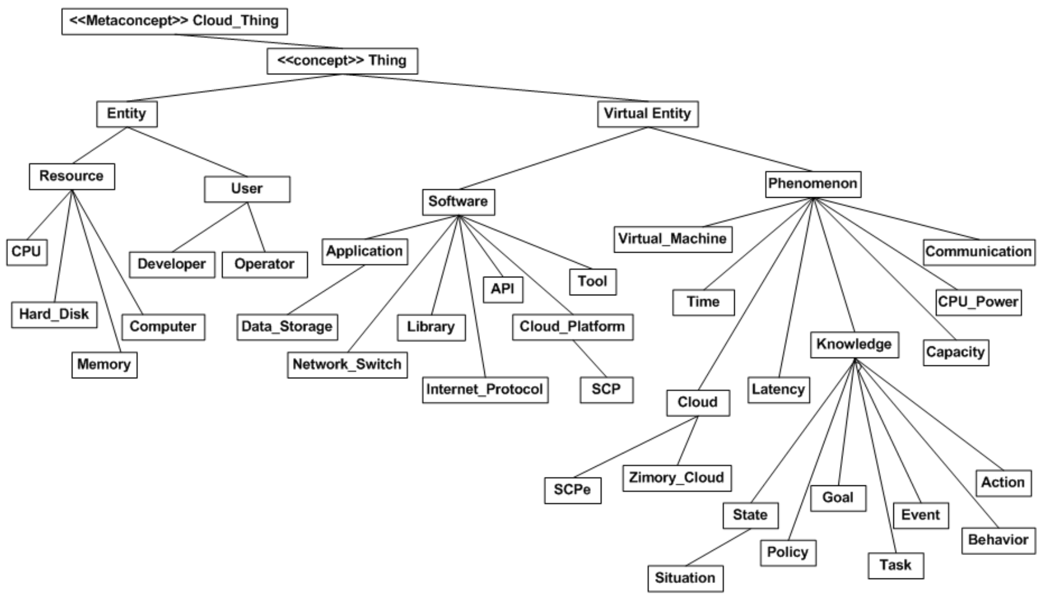


Figure 5: Science Clouds Ontology: Cloud_Thing Concept Tree

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 {          }
    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 [VHM+12] that may handle the *monitoring autonomy requirements* (see Section 3.2).

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

The *Cloud_Thing* concept tree (see Figure 5) is the main concept tree of the SCCloud Ontology. Note that due to space limitations, Figure 5 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 6. As shown, actions are grouped by subsystem (or part) they are associated with. For example, the SCP actions are: *RunSCP*, *StopSCP*, *LeaveSCPe*, and *JoinSCPe*.

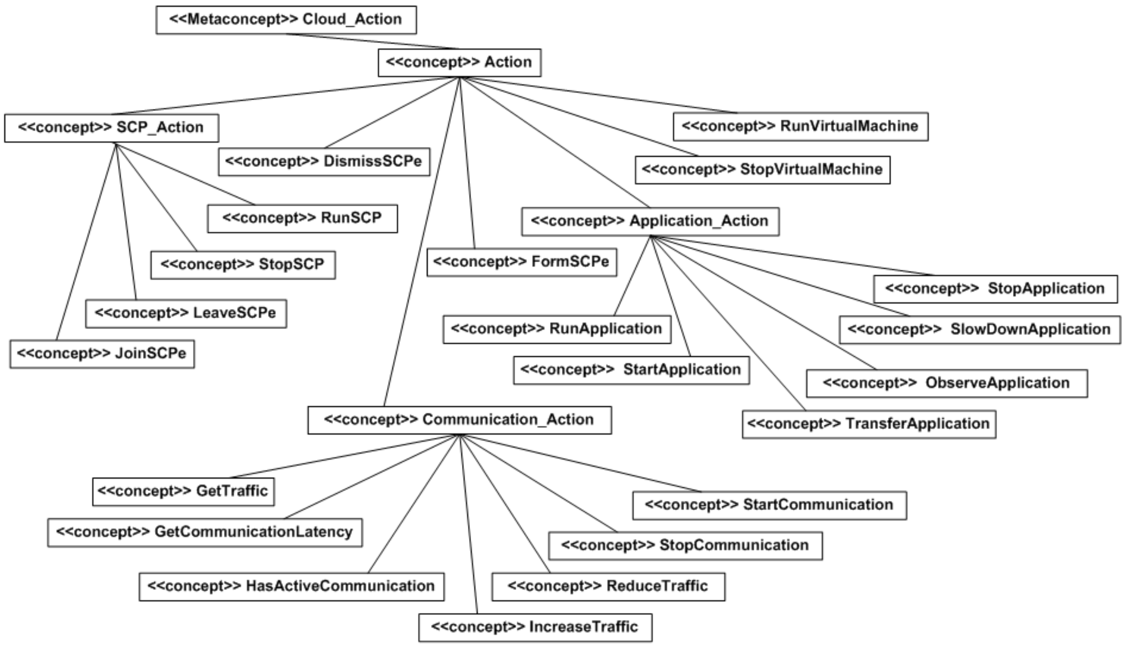

<<Metaconcept>> Cloud_Action
<<concept>> Action
<<concept>> SCP_Action
<<concept>> DismissSCPe
<<concept>> RunVirtualMachine
<<concept>> StopVirtualMachine
<<concept>> RunSCP
<<concept>> StopSCP
<<concept>> LeaveSCPe
<<concept>> JoinSCPe
<<concept>> FormSCPe
<<concept>> Application_Action
<<concept>> StopApplication
<<concept>> SlowDownApplication
<<concept>> RunApplication
<<concept>> StartApplication
<<concept>> ObserveApplication
<<concept>> TransferApplication
<<concept>> Communication_Action
<<concept>> GetTraffic
<<concept>> GetCommunicationLatency
<<concept>> HasActiveCommunication
<<concept>> StartCommunication
<<concept>> StopCommunication
<<concept>> ReduceTraffic
<<concept>> IncreaseTraffic

Figure 6: Science Clouds Ontology: Cloud_Action Concept Tree

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 [VHM+12]. 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, we specified a few exemplar SCPis as object trees, which we do not present here due to space limitations.

3.4.2 Autonomicity

To specify the self-*objectives (autonomicity requirements), we use goals, policies, and situations [VHM+12]. These are defined as explicit concepts in KnowLang and for the Cloud Ontology (SC-

Cloud) we specified them under the concepts *Virtual_entity->Phenomenon->Knowledge* (see Figure 5). Figure 7, 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 3.1) and more specifically, from the goals model for self-* objectives assisting the Science Clouds goals from Level 3 (see Section 3.3).

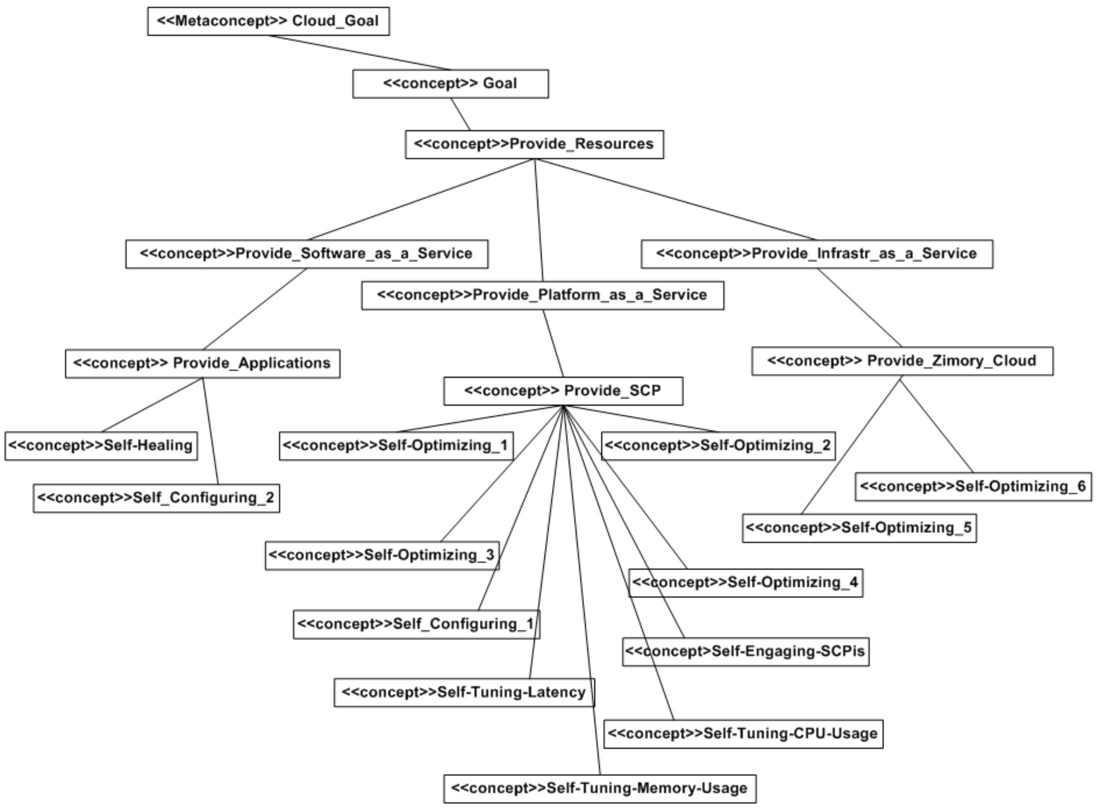

<<Metaconcept>> Cloud_Goal
<<concept>> Goal
<<concept>>Provide_Resources
<<concept>>Provide_Software_as_a_Service
<<concept>>Provide_Platform_as_a_Service
<<concept>>Provide_Infrastr_as_a_Service
<<concept>> Provide_Applications
<<concept>> Provide_SCP
<<concept>> Provide_Zimory_Cloud
<<concept>>Self-Healing
<<concept>>Self_Configuring_2
<<concept>>Self-Optimizing_1
<<concept>>Self-Optimizing_2
<<concept>>Self-Optimizing_6
<<concept>>Self-Optimizing_5
<<concept>>Self-Optimizing_3
<<concept>>Self-Optimizing_4
<<concept>>Self_Configuring_1
<<concept>Self-Engaging-SCPis
<<concept>>Self-Tuning-Latency
<<concept>>Self-Tuning-CPU-Usage
<<concept>>Self-Tuning-Memory-Usage

Figure 7: Science Cloud Ontology: Cloud_Goal Concept Tree

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) [VHM+12]. 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. Recall 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, 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 [VHM+12]. 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 formal presentation of a KnowLang policy specification [VHM+12]. Note that policies (Π) are specified as individual concepts providing behavior (often concurrent). A policy π has a goal (g), policy situations (Si_π), policy-situation relations (R_π), and policy conditions (N_π) mapped to policy actions (A_π) where the evaluation of N_π may eventually (with some degree of probability) imply the evaluation of actions (denoted with $N_\pi \overset{[Z]}{\rightarrow} A_\pi$ (see Definition 2). A condition is a Boolean function over ontology (see Definition 4), e.g., the occurrence of a certain event.

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

Def. 2 $\pi :=< g, Si_\pi, [R_\pi], N_\pi, A_\pi, map(N_\pi, A_\pi, [Z]) >$ *(Policy)*
$A_\pi \subset A, N_\pi \overset{[Z]}{\rightarrow} A_\pi$ *(A_π - Policy Actions)*
$Si_\pi \subset Si, Si_\pi := \{si_{\pi_1}, si_{\pi_2}, ..., si_{\pi_n}\}, n \geq 0$ *(Si_π - Policy Situations)*
$R_\pi \subset R, R_\pi := \{r_{\pi_1}, r_{\pi_2}, ..., r_{\pi_n}\}, n \geq 0$ *(R_π-Policy-Situation Relations)*
$\forall r_\pi \in R_\pi \bullet (r_\pi :=< si_\pi, [rn], [Z], \pi >), si_\pi \in Si_\pi$
$Si_\pi \overset{[R_\pi]}{\rightarrow} \pi \rightarrow N_\pi$ *(Policy situations may imply the policy they are related to)*

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

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

Policy situations (Si_π) are situations that may trigger (or imply) a policy π, in compliance with the policy-situations relations R_π (denoted with $Si_\pi \overset{[R_\pi]}{\rightarrow} \pi$), thus implying the evaluation of the policy conditions N_π (denoted with $\pi \rightarrow N_\pi$)(see Definition 2). Therefore, the optional policy-situation relations (R_π) justify the relationships between a policy and the associated situations (see Definition 2). In addition, the self-adaptive behavior requires relations to be specified to connect policies with situations over an optional probability distribution (Z) where a policy might be related to multiple situations and vice versa. Probability distribution is provided to support probabilistic reasoning and to help the KnowLang Reasoner choose the most probable situation-policy "pair". Thus, we may specify a few relations connecting a specific situation to different policies to be undertaken when the system is in that particular situation and the probability distribution over these relations (involving the same situation) should help the KnowLang Reasoner decide which policy to choose (denoted with $Si_\pi \overset{[R_\pi]}{\rightarrow} \pi$ - see Definition 2). Hence, the presence of probabilistic beliefs at both mappings and policy relations justifies the probability of policy execution, which may vary with time.

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 }
        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 3.3). 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, we 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}
      }
    }
  }
}
```

Recall that situations are related to policies via relations [VHM^{+}12]. The following code demonstrates how we related 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, we 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 [EG05] for probability assessment after action executions [VHM$^+$12].

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 6). 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.

[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).

3.4.3 Monitoring

The *monitoring autonomy requirement* is handled via the explicit *Metric concept* [VHM+12]. In general, a self-adaptive system has sensors that connect it to the world and eventually help it listen to its internal components. These sensors generate raw data that represent the physical characteristics of the world. In our approach, we assume that 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 [VHM+12]:

- *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 3.4.1).

```
//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 } }
  } }
```

4 KnowLang Implementation

The KnowLang Toolset is a comprehensive development environment that delivers a powerful combination of KnowLang notation and KnowLang tools such as Text Editor, Visual Editor, Grammar Compiler, Parser, Consistency Checker and KB Compiler. Currently, we fully implemented the KnowLang's Text Editor, Grammar Compiler and Parser, and we still need to complete the implementation of the Visual Editor, KB Compiler and Consistency Checker.

4.1 KnowLang Toolset

The KnowLang Toolset provides a suitable development environment for knowledge representation (KR) where we can write KR specifications in the KnowLang notation by using both text editing and visual modeling tools and check for the syntactical integrity and consistency of the KR models. The KnowLang Toolset organizes its tools in five distinct components (or modules) such as KnowLang Editor (combines both the Text Editor and Visual Editor), Grammar Compiler, KnowLang Parser, Consistency Checker and KB Compiler. These components are linked together to form a special *KnowLang Specification Processor* that checks and compiles the KR models specified in KnowLang into KnowLang Binary (see Figure 8). As shown, the KnowLang Binary is the output of the KnowLang Toolset and it is practically a *compiled form of the specified KB* (Knowledge Base), which is operated by the KnowLang Reasoner. Note that the KnowLang Reasoner (see Section 4.2) is a distinct KnowLang component intended to be integrated within the systems using knowledge representation with KnowLang, but it also can be integrated in the KnowLang Toolset where it can be used for testing and behavior analysis. Figure 8 presents an abstract view of the KnowLang Toolset

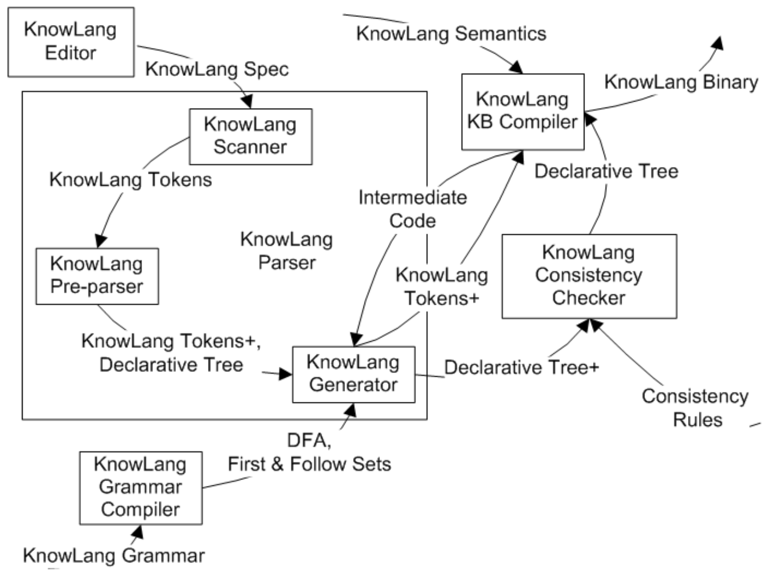


Figure 8: KnowLang Specification Processor: Operational View

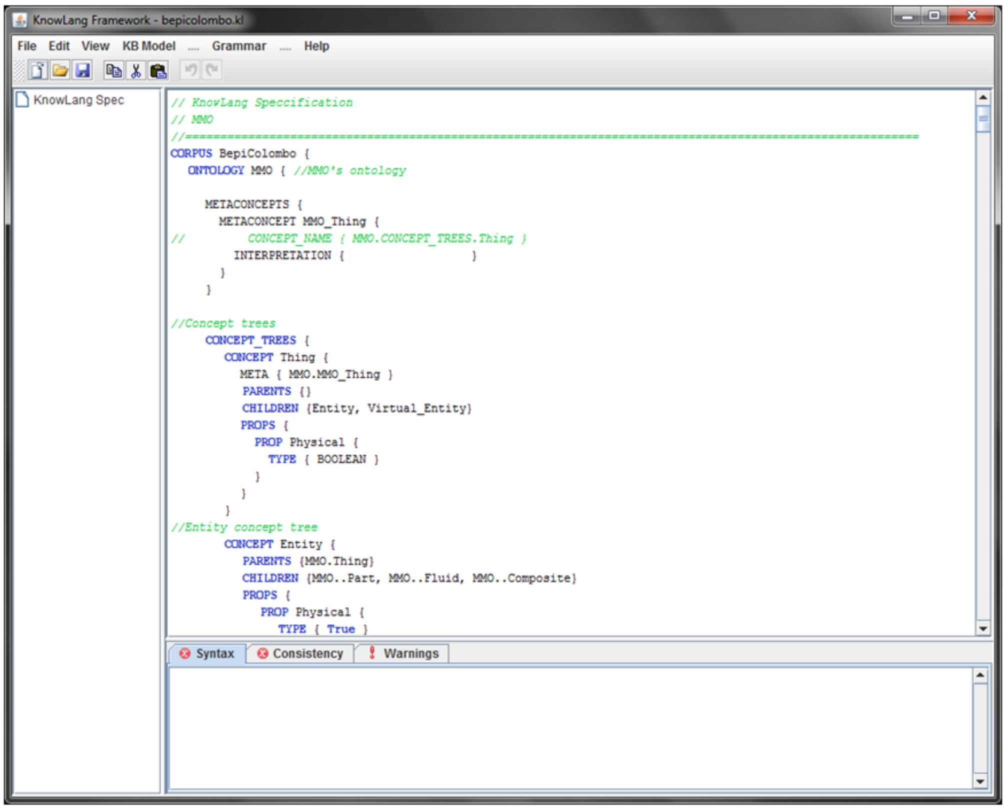
where we can break the operations down by operation type. Thus, we can have groups of components performing the following operations:

- *data source group* (KnowLang Editor + KnowLang Grammar Compiler): prepare the input data (grammar and specification);
- *analysis group* (KnowLang Parser + Consistency Checker): depend on the source language;
- *synthesis group* (KnowLang KB Compiler) depend on the target language (synthesis).

Note that operations that analyze the specification code to compute its properties are classified as *analysis group*, and operations involved in producing the KnowLang Binary are classified as *synthesis group*. Thus, all the operations related to lexical analysis, syntax analysis, and semantic analysis belong to the analysis group, and the generation of the compiled KB is considered synthesis. This kind of operational division shows that the KnowLang Toolset is designed with a view toward changing the output target. This involves implementation of other KB compilers (re-implementation of the synthesis part) for other reasoners (recall that the KnwoLang Binary is intended to be used by the KnowLang Reasoner. Note that although we can also re-implement both analysis and data-source parts of the toolset, this is difficult to achieve without changing the synthesis part, because the latter depends on the intermediate code generated by the analysis part.

4.1.1 KnowLang Editor

The main functions of the KnowLang Editor module (see Figure 9) are to start the KnowLang Framework's Graphical User Interface (GUI) and to load the KnowLang Toolset's structures into memory, including the compiled and serialized KnowLang Grammar's structures. This helps the KnowLang Framework process knowledge models written in KnowLang. The KnowLang Editor module was designed as an application front-end that allows users to interact with the KnowLang Toolset. This module comprises GUI components intended to provide a user-friendly environment to the users. By using these GUI components, the users are able to *load*, *save*, *edit*, and *create* KnowLang specifications in both text and graphical formats. For these operations, the module not only exposes a user-friendly GUI, but also implements the needed functionality. Moreover, the GUI components help users check the consistency of KnowLang specifications and compile (or serialize) in binary format the corresponding KB. In addition, if the KnowLang grammar needs to be recompiled[2], that module exposes the needed GUI. Note that in order to perform these operations, the KnowLang Editor module calls components implementing the needed functionality. The KnowLang Editor module implements and delegates to these modules a *generic error-handling strategy* allowing for handling critical and non-critical specification errors all over the toolset.



```
// KnowLang Speccification
// MMO
//==========================================================================================
CORPUS BepiColombo {
  ONTOLOGY MMO { //MMO's ontology

    METACONCEPTS {
      METACONCEPT MMO_Thing {
//          CONCEPT_NAME { MMO.CONCEPT_TREES.Thing }
        INTERPRETATION {                    }
      }
    }

//Concept trees
    CONCEPT_TREES {
       CONCEPT Thing {
         META { MMO.MMO_Thing }
          PARENTS {}
          CHILDREN {Entity, Virtual_Entity}
          PROPS {
            PROP Physical {
              TYPE { BOOLEAN }
            }
          }
       }
//Entity concept tree
       CONCEPT Entity {
          PARENTS {MMO.Thing}
          CHILDREN {MMO..Part, MMO..Fluid, MMO..Composite}
          PROPS {
             PROP Physical {
               TYPE { True }
```

Figure 9: KnowLang Editor

4.1.2 KnowLang Grammar Compiler

This module includes classes intended to work with the KnowLang grammar. It embeds the functionality and data structures needed to parse the KnowLang grammar and to build internal KnowLang grammar structures needed by the KnowLang Parser module. The KnowLang Grammar module provides a set of logically consistent classes which work cooperatively to:

- compute the First and Follow sets [Lou97] for the KnowLanf BNF grammar [Vas12];

[2]In its future releases the KnowLang framework may possibly be extended with new features, including extending the multi-tier model to include new structures. Grammar recompilation is needed if we modify the KnowLang multi-tier specification model.

- prepare an SLR(1) (**S**imple parsing where the input is processed from **L**eft to right and a **R**ightmost derivation is produced by using **1** symbol of look ahead) [Lou97] parsing table for KnowLang based on the computed First and Follow sets. The grammar input is accepted upon reduction by the production
  KnowLang-Spec $\rightarrow$ ***bof*** *Knowledge-Spec* ***eof***
  where shift-reduce conflicts are resolved in favor of shifting and reduce-reduce conflicts do not appear.

Note that the parsing table built by the KnowLang Grammar module is used by the KnowLang Parser module's classes to parse KnowLang specifications by applying the SLR(1) parsing algorithm as it is described in [Lou97]. Moreover, in the KnowLang Grammar module, we define some KnowLang Grammar definitions like grammar terminals, some specific grammar rule tags, and some generic functions that operate on the KnowLang grammar rules. All of those are needed by both KnowLang Parser and KnowLang KB Compiler (see Figure 8).

4.1.3 KnowLang Parser

The KnowLang Parser module defines classes able to process formal specifications written in KnowLang. Note that the parser creates intermediate data structures that are needed by the KnowLang KB Compiler (see Figure 8) to generate the KnowLang Binary for a KnowLang specification. Figure 10 depicts a high-level UML class diagram representing the KnowLang Parser module. As shown in Figure 10, this module embeds classes implementing functionality related to *scanning*, *pre-parsing* and *parsing* of KnowLang specifications. The main classes defined by this module are designed to provide each one of these functions. The data flow in both KnowLang Parser module and KnowLang Consistency Checker module is as follows.

KnowLang code $\rightarrow$ scanner $\rightarrow$ pre-parser $\rightarrow$ parser $\rightarrow$ consistency checker $\rightarrow$ post-parsing data structure

Here, from the KnowLang specification code, the KnowLang Scanner produces a stream of tokens passed to the KnowLang Pre-Parser, which prepares these tokens to be parsed by the real KnowLang Parser. The output produced by the latter is processed by the KnowLang Consistency Checker. Note that erroneous tokens are identified at each phase of this process. Thus, we have different kinds of errors handled by the scanner, pre-parser, parser and consistency checker. Any one of these can stop the compilation process if errors have been discovered. It is important to mention that the KnowLang parser not only parses the KnowLang specifications, but also generates some intermediate code by using the *KnowLangIntermediateGenerator* class, which is implemented by the KnowLang KB Compiler module (see Figure 10).

There are three token classes defined by this module - *KnowLangToken*, *KnowLangTierToken*, and *KnowLangCodeToken*. The first class is the base KnowLang token class. It is used by the *KnowLangScanner* class to convert a KnowLang specification into a stream of tokens and by other major classes to read and update that stream. In addition, the *KnowLangToken* class is used by the classes *KnowLangScanner*, *KnowLangPreParser*, *KnowLangParser*, and *KnowLangConsistencyChecker* to identify the erroneous tokens operating on the token stream. Thus, the *KnowLangToken* class comprises everything needed to present the words of a KnowLang specification, including the *error* itself (in case the token is erroneous). The *KnowLangTierToken* class is used by the KnowLang Parser module to build a *declarative specification tree* where concepts along with their states, properties and functionalities, objects and relations are added to that tree based on their evaluation. The *KnowLangCodeToken* class is used by the *KnowLangParser* class to generate some intermediate code needed by the KnowLang KB Compiler module.

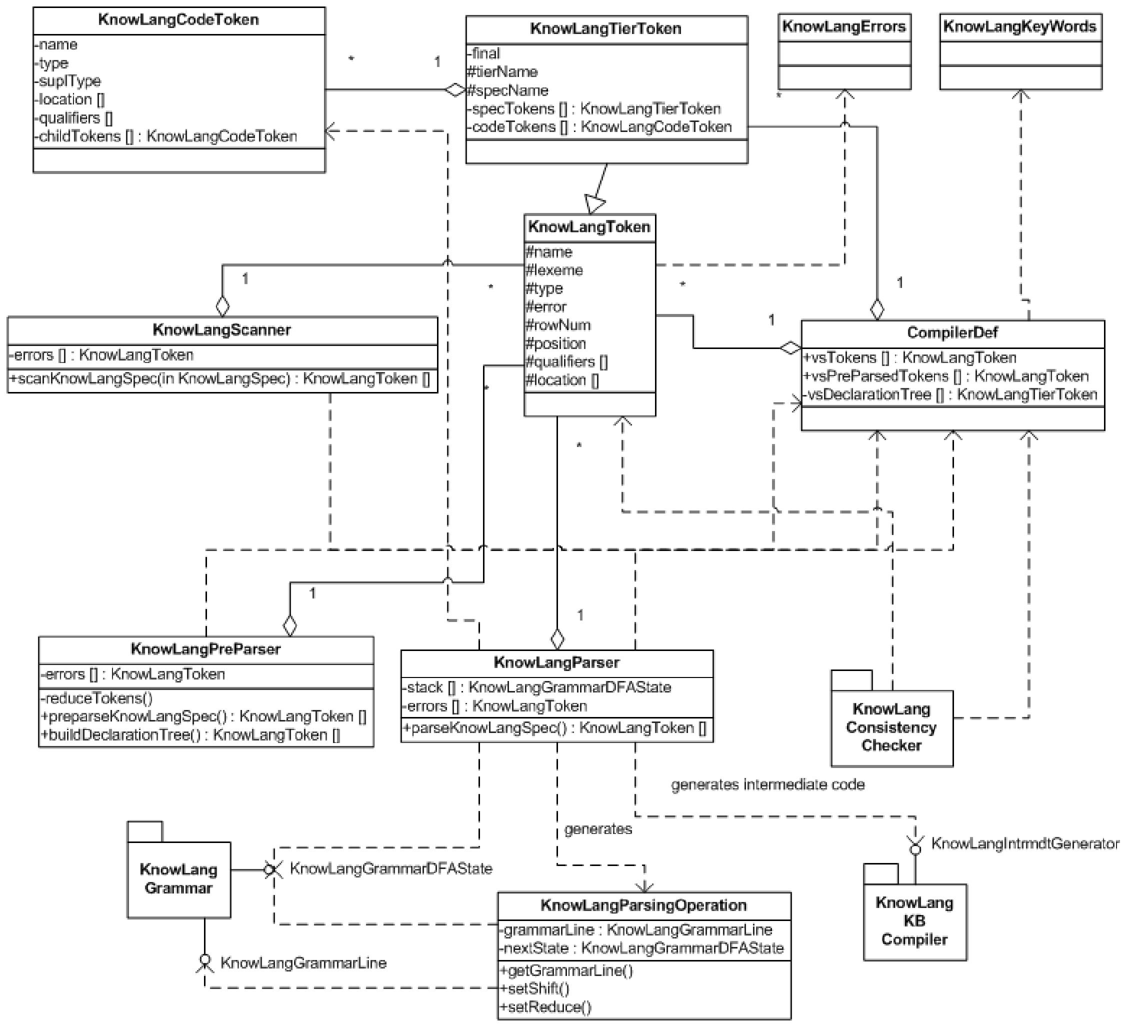

KnowLangCodeToken
-name
-type
-suplType
-location []
-qualifiers []
-childTokens [] : KnowLangCodeToken
KnowLangTierToken
-final
#tierName
#specName
-specTokens [] : KnowLangTierToken
-codeTokens [] : KnowLangCodeToken
KnowLangErrors
KnowLangKeyWords
KnowLangToken
#name
#lexeme
#type
#error
#rowNum
#position
#qualifiers []
#location []
KnowLangScanner
-errors [] : KnowLangToken
+scanKnowLangSpec(in KnowLangSpec) : KnowLangToken []
CompilerDef
+vsTokens [] : KnowLangToken
+vsPreParsedTokens [] : KnowLangToken
-vsDeclarationTree [] : KnowLangTierToken
KnowLangPreParser
-errors [] : KnowLangToken
-reduceTokens()
+preparseKnowLangSpec() : KnowLangToken []
+buildDeclarationTree() : KnowLangToken []
KnowLangParser
-stack [] : KnowLangGrammarDFAState
-errors [] : KnowLangToken
+parseKnowLangSpec() : KnowLangToken []
KnowLang Consistency Checker
generates intermediate code
generates
KnowLangIntrmdtGenerator
KnowLang Grammar
KnowLangGrammarDFAState
KnowLangParsingOperation
-grammarLine : KnowLangGrammarLine
-nextState : KnowLangGrammarDFAState
+getGrammarLine()
+setShift()
+setReduce()
KnowLang KB Compiler
KnowLangGrammarLine

Figure 10: KnowLang Parser UML Class Diagram

The *KnowLangScanner* class performs the so-called *lexical analysis* of the KnowLang specifications, i.e., it collects sequences of characters into meaningful units. The latter emerge as *KnowLangToken* instances in the scanner's output token stream, which is held by the *vsTokens []* array defined by the *CompilerDef* class.

The *KnowLangPreParser* class receives the scanned tokens from the scanner (*KnowLangScanner* class) and performs special *token-consolidation operations* on them (these operations are performed by the *reduceTokens()* method). Some token-consolidation operations are:

- consolidate *RETURNS* and *PARAMETERS* definitions in actions - embed the type in the declaration;
- consolidate the two-symbol Boolean operators (<=, >=) - initially they come from the scanner as a sequence of two tokens each and the pre-parser joins them into one token;
- consolidate qualified names - remove the qualifiers from the token stream and embed them in the token presenting the qualified name (use the *qualifiers []* array declared in the *KnowLangToken* class to embed the qualifiers).

In addition, the pre-parser class constructs the backbone of the so-called *declarative specification tree*. The latter presents the declarative evaluation of a KnowLang specification. Note that the declarative specification tree implies the hierarchical construct of the KnowLang multi-tier specification model.

The *KnowLangParser* class receives the pre-parsed (consolidated) tokens from the pre-parser (*KnowLangPreParser* class) and performs syntax analysis to determine the structure of the KnowLang specification under consideration. In order to perform syntax analysis, the parser uses the SLR(1) parsing table in the form of KnowLang Grammar lines and DFA (deterministic finite automaton) states, these constructed and provided by the KnowLang Grammar module (see Figure 10). While parsing, this class reports any token that does not conform to the KnowLang Grammar rules as being erroneous. The KnowLang Editor module processes the erroneous tokens received by the parser as syntax errors. Moreover, the *KnowLangParser* class uses an instance of the *KnowLangIntrmdtGenerator* class (defined in the KnowLang KB Compiler module) to generate on-the-fly *intermediate code* data in the form of code tokens (instances of the *KnowLangCodeToken* class). This intermediate code is added to the declarative specification tree, which is initially generated by the pre-parser.

4.1.4 KnowLang Consistency Checker

This module uses both the declarative specification tree (constructed by the pre-parser and enriched by the parser with the intermediate code - see Section 4.1.3) and the stream of parsed tokens to check the specification for consistency errors, e.g., double definitions, name reuse, etc.. The KnowLang Consistency Checker performs consistency-checking operations *before* and *after parsing*. In the first group fall lightweight operations, which together with consistency checking do also some special work on the declarative specification tree and on the stream of tokens (coming from the pre-parser). For example, such an operation is checking the consistency of all the identifiers in a KnowLang specification. This involves:

- matching identifiers with their declaration, e.g., an ontology concept or object;
- matching each object with its concept - all objects must be instantiated from a concept specified in an ontology;
- checking for proper use of properties and functions;
- checking if two objects are instantiated from the same concept when compare.

In the second group fall consistency-checking operations, which require that a KnowLang specification be validated by the parser before performing consistency checking on it. It is important to mention, that the KnowLang Consistency Checker generates *consistency errors*. Moreover, it could generate *warnings* while performing consistency-checking operations. Warnings cannot be considered as consistency errors, because they do not contradict the definitions for consistency checking, but rather introduce uncertainty how the KnowLang Reasoner will handle specific cases. This module is still under development.

4.1.5 KnowLang KB Compiler

Basically, this module produces the KnowLang Binary that actually is a serialized version of the represented knowledge. The module takes the declarative tree produced by the KnowLang Parser and checked by the KnowLang Consistency Checker and converts it into a serializable hierarchical structure of concepts and objects. The KnowLang Binary copes with the KnowLang Reasoner, so it can be loaded, queried and updated by that reasoner. This module is still under development and it will be fully completed along with the implementation of the KnowLang Reasoner. This is required, because the KB structure must cope with multiple generic traversal functions for traversing vectors and trees implemented by the KnowLang Reasoner. Generic traversal is achieved through pattern matching.

4.2 KnowLang Resoner

As described in the second deliverable of WP3 [VHM+12], the KnowLang Reasoner is meant to support reasoning about self-adaptive behavior and to provide a KR gateway for the ASK and TELL Operators. The reasoner communicates with the system via ASK and TELL Operators (forming a communication interface) and operates in the KR Context defined by the compiled KB (or KnwoLang Binary), a context formed by the represented knowledge (see Figure 11). The reasoner is compiled as a library (component) that is self-contained and communicates with the world outside via two groups of interfaces defined by the ASK and TELL operators. This allows the reasoner run as a component in the operational context of an ASCENS system hosting that component (see Figure 11). Currently,

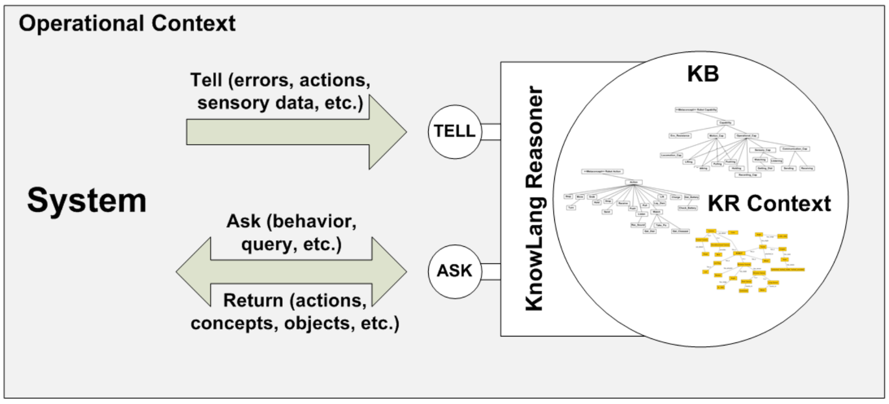

Operational Context
System
Tell (errors, actions, sensory data, etc.)
TELL
Ask (behavior, query, etc.)
Return (actions, concepts, objects, etc.)
ASK
KnowLang Reasoner
KB
KR Context

Figure 11: KnowLang Reasoner

we are working on the implementation of the KnowLang Reasoner where the main effort is on the implementation of the predefined set of ASK and TELL Operators [VHM+12]. Recall that the TELL Operators feed the KR Context with important information driven by errors, executed actions, new sensory data, etc., thus helping the KnowLang Reasoner update the KR with recent changes in both the system and execution environment. The system uses ASK Operators to receive recommended behavior where knowledge is used against the perception of the world to generate appropriate actions in compliance to some goals and beliefs. In addition, ASK Operators may provide the system with awareness-based conclusions about the current state of the system or the environment and ideally with behavior models for self-adaptation. Currently, we are implementing the operational semantics of the following TELL and ASK Operators [VHM+12]:

- TELL_ERR - tells about a raised error;
- TELL_SENSOR - tells about new data collected by a sensor;
- TELL_ACTION - tells about action execution;
- TELL_ACTION (behavior) - tells about action execution as part of behavior performance;
- TELL_OBJ_UPDATE - tells about a possible object update;
- TELL_CNCPT_UPDATE - tells about a possible concept update;

- ASK_BEHAVIOR - asks for self-adaptive behavior considering the current situation;
- ASK_BEHAVIOR(goal) - asks for self-adaptive behavior to achieve certain goal;
- ASK_BEHAVIOR(situation; goal) - asks for self-adaptive behavior to achieve certain goal when departing from a specific situation;
- ASK_BEHAVIOR(state) - asks for self-adaptive behavior to go to a certain state;
- ASK_RULE_BEHAVIOR(conditions) - asks for rule-based behavior;
- ASK_CURR_STATE(object) - asks for the current state of an object;
- ASK_CURR_STATE - asks for the current system state;
- ASK_CURR_SITUATION - asks for the current situation.

Note that each one of these operators is exposed as a public interface of the reasoner and can be directly called by the hosting application.

Another major effort related to the implementation of the KnowLang Reasoner is implementing the awareness mechanism. Here, we are implementing the distinct awareness functions as outlined by the Pyramid of Awareness described in our 2nd WP3 deliverable [VHM+12]: *raw data gathering*, *data passing*, *filtering*, *conversion*, *assessment*, *projection*, and *learning*. The learning functionality is implemented as probability redistribution at the level of policy-situation relations along with probability redistribution at the level of action mapping within a policy definition (see the formal model in Section 3.4.2). Recall that, these functions grouped into four groups from the awareness control loop implemented by the reasoner. The four groups of tasks are: *monitoring tasks*, *recognition tasks*, *assessment tasks*, and *learning tasks*.

Basically, the awareness control loop is implemented to: 1) perform monitoring by analysing the metrics and events; 2) recognize situations, thus based on states evaluation; 3) determine the right polices to act; and 4) redistribute the probability to change the beliefs. The awareness control loop is implemented by following the so-called *super loop architecture* [KP07], which is a design pattern usually implemented as a program structure (e.g., a function) comprising an infinite loop that performs all the tasks of a system. The following pseudocode presents the generic implementation of the *super loop architecture*.

```
while (true) {
  Task1();
  Delay_After_Task1();
  Task2();
  Delay_After_Task2();
  ....
  TaskN();
  Delay_After_TaskN();
}
```

As shown, the tasks are performed in a deterministic order with some delays between them. These delays are optional and are intended to keep the execution of tasks within a time frame allocated for each task. Here, the delays should be computed dynamically at runtime by considering the last execution time of each task for each loop pass. Note that task timing is important to guarantee that the reasoned will act efficiently and meet eventual time deadlines if such exist. Thus, this architecture targets at performing all the tasks in a correct deterministic sequential order and possibly in a reasonable amount of time.

Moreover, this architecture helps us realize different levels of awareness exhibition and eventually degree of awareness [VHM+12]. Recall that the levels of awareness might be related to data readability and reliability, i.e., it could happen to have noisy data that must be cleaned up, which may slow down the whole process and to keep up with the assigned task deadline the reasoner will stop the process at certain point and data will be eventually interpreted with some degree of probability. Moreover, this loop architecture helps us realize other levels of awareness exhibition such as early awareness, which is supposed to be a product of one or two passes of the awareness control loop and late awareness, which should be more mature in terms of conclusions and projections. Therefore, similar to humans who may react to their first impression and then the reaction might shift together with a late but better realization of the current situation, the KnowLang Reasoner relies on early awareness to react quickly to situations when fast reaction is needed and on late awareness when more precise thinking is required.

5 Summary and Future Goals

In the course of the third year of WP3, we continued working on the KnowLang Framework implementation along with improving the efficiency in knowledge representation with KnowLang. To find the right level of abstraction when specifying knowledge with KnowLang, in a joint project with ESA, the European Space Agency, we developed an approach to Autonomy Requirements Engineering (ARE). Consecutively, we used the ARE approach to build efficient and relevant knowledge models for ASCENS. With ARE, we introduced a preliminary software engineering step to the process of Knowledge Engineering. This step helped us select and refine relevant and efficient knowledge data that we knowledge-represented with KnowLang for the ASCENS Science Clouds case study. The approach focuses on the so-called self-* objectives, providing for self-adaptive behavior, and consecutively centers the knowledge models around this self-adaptive behavior. This makes the knowledge representation very efficient and specified at the right level of abstraction. Note that this was a major flaw in our previous knowledge models, which carried unnecessary details, thus eventually overwhelming the reasoning process.

In this 3rd year of the project, we also started implementing the KnowLang Reasoner where the main efforts were on the implementation of the ASK and TELL Operators along with the awareness control loop. Currently, we are implementing the operational semantics of these operators. As for the awareness control loop, we are using a super loop architecture that helps us realize different levels of awareness exhibition and eventually degree of awareness. Basically, this architecture introduces a deadline to each one of the awareness loop tasks and the levels of awareness exhibition are a product of different number of loop iterations.

Our plans for the fourth year of WP3 are mainly concerned with further development of the KnowLang Reasoner, to finish up Task 3.3. Along with this task, we shall implement the awareness prototypes (Task 3.4) based on the new knowledge representation models for the ASCENS case studies (Task 3.2) developed with the ARE approach. This requires that we build complete knowledge models by applying the ARE approach to the other two case studies, as we did for the Science Clouds case study. Along with the completion of this task, we need to work in close collaboration with WP4 on the full integration of the ARE approach and SOTA. Last but not least we need to complete the KnowLang Framework's modules, which are still not fully implemented as described in Section 4.

References

[EG05] W. Ewens and G. Grant. Stochastic processes (i): Poison processes and Markov chains. In *Statistical Methods in Bioinformatics, 2nd edition, Springer, New York*, 2005.

[HK13] M. Hölzl and N. Koch. D8.3: Third Report on WP8: Best Practices for SCEs, 2013. ASCENS Deliverable.

[HPW98] P. Haumer, K. Pohl, and K. Weidenhaupt. Requirements elicitation and validation with real world scenes. *IEEE Transactions on Software Engineering - Special Issue on Scenario Management*, pages 1036–1054, 1998.

[KHK$^+$13] N. Koch, M. Hölzl, A. Klarl, P. Mayer, T. Bures, J. Combaz, A.L. Lafuente, R. De Nicola, S. Sebastio, F. Tiezzi, A. Vandin, F. Gaducci, U. Montanari, M. Loreti, C. Pinciroli, M. Puviani, F. Zambonelli, N. Serbedzija, and E. Vassev. JD3.2: Software Engineering for Self-Aware SCEs: Ensemble Development Life Cycle, 2013. ASCENS Deliverable.

[KP07] S. Kurian and M.J. Pont. Maintenance and evolution of resource-constrained embedded systems created using design patterns. *Journal of Systems and Software*, 80(1):32–41, 2007.

[Lou97] K. C. Louden. *Compiler Construction - Principles and Practice*. PWS, Boston, MA, USA, 1997.

[MKH$^+$13] P. Mayer, A. Klarl, R. Hennicker, M. Puviani, F. Tiezzi, R. Pugliese, J. Keznikl, and T. Bures. 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*, pages 1–6, Philadelphia, USA, September 2013.

[PNAZ13] M. Puviani, V. Noel, D. Abeywickrama, and Franco Zambonelli. D4.3: Third Report on WP4, 2013. ASCENS Deliverable.

[RS77] D.T. Ross and K.E. Schoman. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, 3(1):6–15, 1977.

[RSA98] C. Rolland, C. Souveyet, and C.B. Achour. Guiding goal-modeling using scenarios. *IEEE Transactions on Software Engineering - Special Issue on Scenario Management*, pages 1055–1071, 1998.

[SHP$^+$13] N. Serbedzija, N. Hoch, C. Pinciroli, M. Kit, T. Bures, G.V. Monreale, U. Montanari, P. Mayer, and J. Velasco. D7.3: Third Report on WP7: Integration and Simulation Report for the ASCENS Case Studies, 2013. ASCENS Deliverable.

[SMP$^+$12] N. Serbedzija, M. Massink, C. Pinciroli, M. Brambilla, D. Latella, M. Dorigo, M. Birattari, P. Mayer, J.A. Velasco, N. Hoch, H.P. Bensler, D. Abeywickrama, J. Keznikl, I. Gerostathopoulos, T. Bures, R. De Nicola, and M. Loreti. D7.2: Second Report on WP7: Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility, 2012. ASCENS Deliverable.

[SRA$^+$11] N. Serbedzija, S. Reiter, M. Ahrens, J. Velasco, C. Pinciroli, N. Hoch, and B. Werther. D7.1: First Report on WP7: Requirement Specification and Scenario Description of the ASCENS Case Studies, 2011. ASCENS Deliverable.

[Vas12] E. Vassev. KnowLang grammar in BNF. Technical Report Lero-TR-2012-04, Lero, University of Limerick, Ireland, 2012.

[VH13a] E. Vassev and M. Hinchey. Autonomy requirements engineering. In *Proceedings of the 14th IEEE International Conference on Information Reuse and Integration (IRI'13)*, pages 175–184. IEEE Computer Society, 2013.

[VH13b] E. Vassev and M. Hinchey. Autonomy requirements engineering. *IEEE Computer*, 46(8):82–84, 2013.

[VH13c] E. Vassev and M. Hinchey. Autonomy requirements engineering: A case study on the BepiColombo Mission. In *Proceedings of the C* Conference on Computer Science & Software Engineering (C3S2E'13)*, pages 31–41. ACM, 2013.

[VH13d] E. Vassev and M. Hinchey. 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, 2013.

[VHM$^+$12] E. Vassev, M. Hinchey, U. Montanari, N. Bicocchi, F. Zambonelli, and M. Wirsing. D3.2: Second Report on WP3: The KnowLang Framework for Knowledge Modeling for SCE Systems, 2012. ASCENS Deliverable.

[vL00] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'2000)*, pages 5–19. ACM, 2000.

[vLDM95] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In *Proceedings of the 2nd International IEEE Symposium on Requirements Engineering*, pages 194–203. IEEE, 1995.