www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D3.2: Second Report on WP3

The KnowLang Framework for Knowledge Modeling for SCE Systems

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), Ugo Montanari (UNIPI), Nicola Bicocchi (UNIMORE), Franco Zambonelli (UNIMORE), Martin Wirsing (LMU)**

Reporting Period: **2**
Period covered: **October 1, 2011 to September 30, 2012**
Submission date: **November 12, 2012**
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. 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 second year of WP3, to maximize the impact of our work, we focused on the research and development of KnowLang and a proper awareness mechanism. The KnowLang specification model has gradually evolved over the last year to take a more mature shape. This helped us to specify initial knowledge models for all three ASCENS case studies. In addition, we started working on the KnowLang Reasoner and started implementing the KnowLang Toolset. An important break-through is the KnowLang mechanism for self-adaptive behavior where knowledge representation and reasoning help to establish the vital connection between knowledge, perception and actions realizing self-adaptive behavior. To support this approach, we developed a KR mechanism for self-adaptive behavior and started working on special ASK and TELL operators used by the system to talk to the KnowLang Reasoner. Moreover, we developed a conceptual reference model for awareness called ”Pyramid of Awareness” and outlined how this model can be realized with the KnowLang Framework. Finally, to allow for knowledge representation of liveness properties, we started working on a possible integration of soft constraints in KnowLang. Note that our work on KnowLang required intensive collaboration with WP1, WP2, WP4 and WP7 and gradual integration of KnowLang with SCEL and SOTA tackled by WP1 and WP4 respectively.

Contents

1 Introduction

Contemporary computerized systems like autonomous robots may boast intrinsic intelligence that helps them reason about situations where autonomous decision making is required. Robotic intelligence mainly excels at formal logic, which allows it, for example, to find the right move from hundreds of previous moves or by applying probability algorithms. The basic compound in this reasoning process is appropriately structured knowledge used by embedded inference engines. The knowledge is integrated in a system via knowledge representation techniques to build a computational model of the operational domain in which symbols serve as knowledge surrogates for real world artefacts, such as system's components and functions, task details, environment objects, etc. The domain of interest can cover any part of the real world or any hypothetical system about which one desires to represent knowledge for computational purposes. Knowledge representation primitives such as rules, frames, semantic networks, concept maps, ontologies, and logic expressions might be used to represent distinct pieces of knowledge that are worth being differently represented. Moreover, these primitives might be combined into more complex knowledge elements. Whatever elements they use, engineers must structure the knowledge so that the system can effectively process it and eventually derive its own behavior.

1.1 Research Focus

One of the main scientific contributions that we expect to achieve with ASCENS is related to Knowledge Representation and Reasoning (KR&R). Within the WP3's mandate of the project, we are currently developing the KnowLang Framework that offers a notion for modeling knowledge and self-adaptive behavior of ASCENS-like systems.

In this second year of WP3, without changing the overall goals of WP3, we reorganized the tasks to maximize the impact of our work. Thus, we focused our Research and Development (R&D) on the development of KnowLang and a proper awareness mechanism. Although still under development, the *KnowLang Specification Model* has gradually evolved over the last year of the project by taking a more mature shape. We used KnowLang to specify some initial knowledge models for all three ASCENS case studies.

In addition, along with further development of the language theory and knowledge-specification structures, we started working on the KnowLang Reasoner and started implementing the KnowLang Toolset. An important break-through is the KnowLang *mechanism for self-adaptive behavior* where knowledge representation and reasoning help to establish the vital connection between knowledge, perception and actions realizing self-adaptive behavior. The knowledge is used against the perception of the world to generate appropriate actions in compliance to some goals and beliefs.

To support this approach, we developed a KR mechanism for self-adaptive behavior and started working on special ASK and TELL operators used by the system to talk to the KnowLang Reasoner. We developed an initial operational semantics for these operators. Moreover, we developed a conceptual reference model for awareness called ”Pyramid of Awareness” and outlined how this model can be realized with the KnowLang Framework.

In collaboration with WP2, to allow for knowledge representation of liveness properties, we started working on a possible integration of soft constraints with KnowLang. Moreover, we started a gradual integration of KnowLang with SCEL and SOTA, tackled by WP1 and WP4 respectively, and continued working on the KR models for all three case studies supported by WP7.

1.2 Tasks Shifting

As originally reported, there are four tasks to be completed in WP3. In the course of this second year, by following the KnowLang's trend of R&D, we had to focus the first three tasks more on KnowLang as following:

- Task 1. This task is entirely dedicated to KnowLang now. So far, this task has been consuming most of the R&D time of WP3 and is mainly focussed on KnowLang. As initially described, the task also had to carry the development of knowledge models. This required focus shifting and to avoid that we moved this development to Task 2.
- Task 2. Following the change in Task 1, Task 2 has been shifted towards creating basic KR Models for the ASCEN's Case Studies. Along with the implementation of the language, we started specifying basic KR models for all the three ASCENS case studies. The complexity of the problem stemming from the large diversity of the case studies and the R&D of KnowLang helped us to realize that we need extra effort on that. Moreover, the so-called Generic Knowledge Models, which were initially intended to be developed first, appeared to be better developed by merging knowledge models of the ASCENS case studies.
- Task 3. This task is now dedicated to awareness and reasoning. The original task was more general and was supposed to carry research and investigation on techniques for knowledge processing and update. However, the narrowed R&D on KnowLang helped us to make this task more focused and driven by Task 1. This task is currently tackling the problem of developing the KnowLang Reasoner, capable of deducting self-adaptive behavior, along with the reference model for self-awareness. A major part of the activities in the task shall be consolidated around the special ASK and TELL operators introduced by KnowLang for communication with the KnowLang Reasoner.

Note that, currently, Task 4 has no changes to carry out, and as shaped, all the tasks shall help us better achieve the ultimate goals of WP3, i.e., to develop a KR method and reasoning mechanism for self-awareness in ASCENS systems.

1.3 Relations with Other WPs

In this second year of the project, we started collaborating more intensively with WP1, WP2, WP4 and WP7. Note that our work on the KnowLang KR&R mechanism and awareness required strong collaboration with WP1 and WP4 (see Section 6). Note that KnowLang provides a KR model of the SCEL (tackled by WP1) knowledge base and the Knowlang Reasoner should be properly integrated with SCEL (via the ASK and TELL operators). Considering WP4, KnowLang will be used to model situations and self-adaptation policies determined with SOTA, the State Of The Affairs framework tackled by WP4. Moreover, WP4 has compiled an extensive catalogue of adaptation patterns, which we used to derive some self-adaptation scenarios for the marXbot case study.

In collaboration with WP2, we developed a theoretical model for integrating the so-called soft constraints with KnowLang (see Section 5). 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.

WP7 provides vital experimental platforms for both the notation and toolset of KnowLang. Thus, in collaboration with WP7, we used KnowLang to specify some initial knowledge models for all three ASCENS case studies (see Section 3). The most explored case study from our side was the one of marXbot robotics platform. For this case study, along with the intensive specification of initial knowledge models (ontologies, facts, rules and constraints), we also specified behavior models (although still theoretical) and played scenarios with the same (see Section 2.4.2). For the other two case studies, we specified initial knowledge models based on their case studies, which however, still need further refinement at the level of detailed knowledge. For example, for the Science Cloud case study, we developed an initial model for the SC ontology by specifying basic concept trees such as Science Cloud Thing, Property, Quality and Information Structure. These concept trees represent the basic concepts in the domain outlined by the Science Cloud SC terminology. For example, the Information Structure concept tree hierarchically relates the basic concepts used to classify information-structuring mechanisms such as File, Query, DB Table, List, Stack, Queue, etc. The main challenge with all these knowledge models is to identify the right level of abstraction at which reasoning can provide for adaptation and self-awareness. Hence, our initial idea to construct ”generic knowledge models” first and then derive from those the case-study-specific knowledge models turned around to take a shape where the specific models will be developed first and the generic models will be derived by overlapping the models for all three case studies.

1.4 Document Organization

The rest of this document is organized as follows. Section 2 covers in detail the current state of the KnowLang Framework in terms of specification model, challenges, syntax, reasoner and toolset. Section 3 presents KR models created with KnowLang for the ASCENS case studies. In Section 4, we present our research on the Pyramid of Awareness and in Section 5, we discuss the integration of soft constraints with KnowLang. Finally, to conclude the topic, in Section 6, we discuss the relation of KnowLang with SCEL and SOTA and present a brief summary and future goals in Section 7.

2 KnowLang

KnowLang [VHng, VH12c, VHG12, VH12d, VH11, VH12a] is a framework for KR&R that aims at efficient and comprehensive knowledge structuring and awareness based on *logical* and *statistical reasoning*. It helps us to tackle 1) explicit representation of domain concepts and relationships; 2) explicit representation of particular and general factual knowledge, in terms of predicates, names, connectives, quantifiers and identity; and 3) uncertain knowledge in which additive probabilities are used to represent degrees of belief. Other remarkable features are related to knowledge cleaning (allowing for efficient reasoning) and knowledge representation for autonomic self-adaptive behavior. Knowledge specified with KnowLang takes the form of a Knowledge Base (KB) that outlines a KR context. A special KnowLang Reasoner operates in this context to allow for knowledge querying and update. In addition, the reasoner can infer special self-adaptive behavior.

2.1 Specification Model

At its very core, KnowLang is a formal specification language providing a comprehensive specification model aiming at addressing the knowledge representation problem for ASCENS-like systems. The complexity of the problem necessitated the use of a *specification model* (inspired by the ASSL's specification model [Vas09]) where knowledge can be presented at different levels of abstraction and grouped by following both hierarchical and functional patterns. KnowLang imposes a multi-tier specification model (see Figure 1), where we specify a KB composed of layers dedicated to *knowledge corpuses*, *KB (knowledge base) operators* and *inference primitives*.

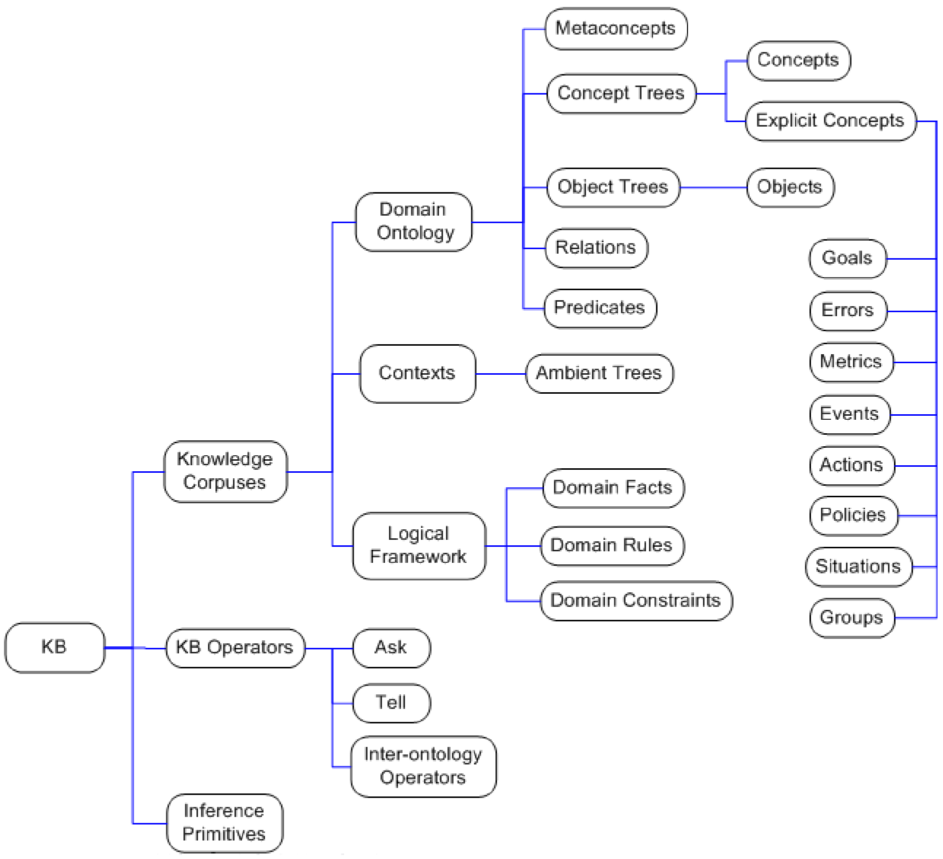


Figure 1: KnowLang Specification Model

Definitions 1 through 58 outline a BNF-like [Knu64] formal representation of the KnowLang Specification Model. As shown in Definition 1, a Knowledge Base is a tuple of three main knowledge components - *knowledge corpus* (Kc), *KB operators* (Op) and *inference primitives* (Ip). A Kc is a tuple of three knowledge components - *ontologies* (O), *contexts* (Cx) and *logical framework* (Lf) (see Definition 2). Further, a domain ontology is composed of hierarchically organized sets of *meta-concepts* (Cm), *concept trees* (Ct), *object trees* (Ot), *relations* (R) and *predicates* (V) (see Definition 4). Note that the trees in our model (e.g., concept trees, object trees, etc.) can be direct acyclic graphs. Moreover, note that in the definitions below we denote a finite set of elements El with $\{el_1, el_2, ..., el_n\}, n \geq 0$ where by omitting el_0 we allow an empty set, e.g., see the definition of meta-concepts (Cm) 5.

Meta-concepts (Cm) provide a *context-oriented interpretation* (i) (see Definition 6) of concepts and might be optionally associated with specific contexts (the square brackets ”[]” mean ”optional”). Meta-concepts help ontologies to be viewed from different context perspectives by establishing different meanings for some of the key concepts. This is a powerful construct providing for interpretations of a concept and its derived concept tree depending on the current context. Concept trees (Ct) consist of semantically related concepts (C) and/or explicit concepts (Ce). Every concept tree (ct) has a root concept (tr) because the architecture ultimately must reference a single concept that is the connection point to concepts that are outside the concept tree. A root concept may optionally inherit a meta-concept, which is denoted [$tr \succ cm$] (see Definition 8) where ”$\succ$” is the inherits relation. Every concept has a set of *properties* (P) and optional sets of *functionalities* (F), *parent concepts* (Pr) and *children concepts* (Ch) (see Definition 10). Explicit concepts are concepts that *must be presented* in the KB of the system. Explicit concepts are mainly intended to support 1) the autonomic behavior of the SCs; and 2) distributed reasoning and knowledge sharing among the SC of a SCE systems. These concepts might be *goals* (G), *errors* (Er), *metrics* (M), *policies* (Π), *events* (E), *actions* (A), *situations* (Si) and *groups* (Gr) (see Definition 13), i.e., they allow for quantification over such concepts.

FORMAL REPRESENTATION OF KNOWLANG

Def. 1 $Kb :=< Kc, Op, Ip >$ *(Knowledge Base)*

Def. 2 $Kc :=< O, Cx, Lf >$ *(Knowledge Corpus)*

DOMAIN ONTOLOGIES

Def. 3 $O := \{o_{sc}, o_{sce}, o_{env}, o_{si}\}$ *(Domain Ontologies)*

Def. 4 $o :=< Cm, Ct, Ot, R, D >, o \in O$ *(Domain Ontology)*

Def. 5 $Cm := \{cm_1, cm_2, ..., cm_n\}, n \geq 0$ *(Meta-concepts)*

Def. 6 $cm :=< [cx], i >, i \in Icx$ *(Meta-concept, cx - Context, i - Interpretation)*

Def. 7 $Ct := \{ct_1, ct_2, ..., ct_n\}, n \geq 0$ *(Concept Trees)*

Def. 8 $ct :=< tr, C, [Ce] >$ *(Concept Tree)*
$tr \in (C \cup Ce), [tr \succ cm]$ *(tr - Tree Root)*

Def. 9 $C := \{c_1, c_2, ..., c_n\}, n \geq 0$ *(Concepts)*

Def. 10 $c :=< P, [F], [S], [Pr], [Ch] >$ *(Concept)*
$Pr \subset (C \cup Ce), c \succ Pr$ *(Pr - Parents)*
$Ch \subset (C \cup Ce), Ch \succ c$ *(Ch - Children)*

Def. 11 $P := \{p_1, p_2, ..., p_n\}, n \geq 0$ *(Properties)*

Def. 12 $F := \{f_1, f_2, ..., f_n\}, n \geq 0$ *(Functionalities)*

Def. 13 $Ce := G \bigcup Er \bigcup M \bigcup \Pi \bigcup E \bigcup A \bigcup Si \bigcup Gr$ *(Explicit Concepts)*

Errors (Er) are explicit concepts representing the *space of errors* that can occur in the system. An error (er) is specified with *error information* (i_{er}) and an optional set of *erroneous actions* (A_{er}) that could be considered as eventual sources of error (see Definition 15). Error occurrence can cause a state transition (see Definition 22). Metrics (M) are explicit concepts providing a *prognostic space* of valuable information that can be gathered from the environment or from the system itself. A metric (m) is specified with a metric source (sr_m) and data (d_m)(see Definition 17). The metric source may eventually represent a system sensor used to monitor the environment.

Def. 14 $Er := \{er_1, er_2, ..., er_n\}, n \geq 0$ *(Errors)*

Def. 15 $er :=< i_{er}, [A_{er}] >$ *(Error)*
$A_{er} \subset A$ *(A_{er} - Erroneous Actions)*

Def. 16 $M := \{m_1, m_2, ..., m_n\}, n \geq 0$ *(Metrics)*

Def. 17 $m :=< sr_m, d_m >$ *(Metric)(sr_m - Metric Source, d_m - Metric Data)*

The KnowLang policies (Π) drive the autonomic behavior of the system. 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 $N_\pi \overset{[Z]}{\to} A_\pi$) (see Definition 19).

A condition is a Boolean expression over ontology (see Definition 21), e.g., the occurrence of a certain event. *Policy situations* Si_π are situations (see Definition 25) that may trigger (or imply) a policy π, in compliance with the policy-situations relations R_π(denoted with $Si_\pi \overset{[R_\pi]}{\to} \pi$), thus implying the evaluation of the policy conditions N_π(denoted with $\pi \to N_\pi$)(see Definition 19). A policy may comprise optional policy-situation relations (R_π) justifying the relationships between a policy and the associated situations. The presence of probabilistic beliefs in both *mappings* and *policy relations* justifies the probability of policy execution, which may vary with time. Note that Section 2.4.2 discusses in detail how the KR of policies, situations and relations provides for self-adaptive behavior.

A goal g is a desirable transition ($\Rightarrow$) to a state or a transition from a specific state to another state (denoted with $s \Rightarrow s'$) (see Definition 22). The system may transit ($\Rightarrow$) to a state (s) when the properties (P) of an object (ob) are updated (denoted $TELL \rhd ob.P$), the properties of a set of objects are updated, or some errors or events have occurred or actions have been realized in the system or in the environment (denoted with $TELL \rhd Er_s$, $TELL \rhd E_s$ and $TELL \rhd A_s$) (see Definition 22). Note that $TELL$ is a KB Operator involving knowledge inference (see Section 2.4.1). In KnowLang, a state s is a Boolean expression over ontology ($be(O)$)(see Definition 23), e.g., "a specific property of an object must hold a specific value".

A situation is expressed with a state (s), a history of actions ($A \overset{\leftarrow}{_{si}}$) (actions executed to get to state s), actions A_{si} that can be performed from state s and an optional history of events $E \overset{\leftarrow}{_{si}}$ that eventually occurred to get to state s (see Definition 25).

Def. 18 $\Pi := \{\pi_1, \pi_2, ..., \pi_n\}, n \geq 0$ *(Policies)*

Def. 19 $\pi :=< g, Si_\pi, [R_\pi], N_\pi, A_\pi, map(N_\pi, A_\pi, [Z]) >$ *(Policy)*
$A_\pi \subset A, N_\pi \overset{[Z]}{\to} A_\pi$ *(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)*
$\forall r_\pi \in R_\pi \bullet (r_\pi :=< si_\pi, [rn], [Z], \pi >), si_\pi \in Si_\pi$
$Si_\pi \overset{[R_\pi]}{\to} \pi \to N_\pi$ *(Policy situations may imply the policy they are related to)*

Def. 20 $N_\pi := \{n_1, n_2, ..., n_k\}, k \geq 0$ *(Policy Conditions)*

Def. 21 $n := be(O)$ *(Condition - Boolean Expression over Ontology)*

Def. 22 $g := \langle \Rightarrow s' \rangle | \langle s \Rightarrow s' \rangle$ *(Goal)*
$\Rightarrow s := \langle TELL \rhd ob.P \rangle | \langle TELL \rhd \{ob_0.P, ob_1.P, ..., ob_n.P\} \rangle | \langle TELL \rhd Er_s \rangle |$
$\langle TELL \rhd E_s \rangle | \langle TELL \rhd A_s \rangle$ *(State Transition)*
$Er_s \subset Er$, $E_s \subset E$, $A_s \subset A$ *(Er_s - State Errors, E_s - State Events, A_s - State Actions)*

Def. 23 $s := be(O)$ *(State - Boolean Expression over Ontology)*

Def. 24 $Si := \{si_1, si_2, ..., si_n\}, n \geq 0$ *(Situations)*

Def. 25 $si :=< s, A_{\overleftarrow{si}}, [E_{\overleftarrow{si}}], A_{si} >$ *(Situation)*
$A_{\overleftarrow{si}} \subset A$ *($A_{\overleftarrow{si}}$ - Executed Actions)*
$A_{si} \subset A$ *(A_{si} - Possible Actions)*
$E_{\overleftarrow{si}} \subset E$ *($E_{\overleftarrow{si}}$ - Situation Events)*

KnowLang events (E) are a means of high-priority monitoring and messaging. In general, an event (see Definition 27) can be activated (raised) by a variety of factors such as time (t_e), goals (G_e), metrics (M_e), errors (Er_e), actions (A_e) and even other events (E_e). A special *guard* (gd_e), represented as a Boolean expression over ontology (see Definition 28), may restrict the event activation. Events may participate in Boolean expressions or be used to specify event-driven policies, goals, situations, etc.

In KnowLang, actions are activities (routines) that can be performed by the system. Actions must be implemented by the system and with KR we represent an abstraction (counterparts) of the routines and classes used to implement these actions. Therefore, an action concept must refer to real implementation. From KR perspective, an action a is a tuple of optional pre- (rc_a), and post-conditions (pc_a), a set of parameters (Pm_a), output (rn_a) and errors (Er_a) that can be raised by the action (see Definition 30).

Def. 26 $E := \{e_1, e_2, ..., e_n\}, n \geq 0$ *(Events)*

Def. 27 $e :=< [gd_e], activ >$ *(Event)*
$activ := t_e|G_e|M_e|Er_e|A_e|E_e$ *(Activation Factor)*
$G_e \subset G, M_e \subset M, Er_e \subset Er, A_e \subset A, E_e \subset E$

Def. 28 $gd_e := be(O)$ *(Event Guard)*

Def. 29 $A := \{a_1, a_2, ..., a_n\}, n \geq 0$ *(Actions)*

Def. 30 $a :=< [rc_a], [pc_a], [Pm_a], [rn_a], [Er_a] >$ *(Action)*

A group (gr) involves objects (Ob_{gr}) related to each other through a distinct set of relations (R_{gr})(see Definition 32). Note that groups (G) are explicit concepts intended to (but not restricted to) represent knowledge about the structure of the system.

Object trees (Ot) are conceptualization of how objects existing in the world of interest are related to each other. The relationships are based on the principle that objects have properties, where sometimes the value of a property is another object, which in turn also has properties. Such properties are termed object properties (Pb). An object tree (ot) consists of a root object (ob) and an optional set of object properties (Pb) - sub-trees of objects (see Definitions 34 and 36). An object (ob) is an instance of a concept (denoted as $instof(c)$ - see Definition 35) and inherits that concept's properties.

Def. 31 $Gr := \{gr_1, gr_2, ..., gr_n\}, n \geq 0$ *(Groups)*

Def. 32 $gr :=< Ob_{gr}, R_{gr} >$ *(Group)*
$Ob_{gr} \subset Ob, R_{gr} \subset R$ *(Ob_{gr}-Group Objects, Ob - Objects, R_{gr}-Group Relations)*

Def. 33 $Ot := \{ot_1, ot_2, ..., ot_n\}, n \geq 0$ *(Object Trees)*

Def. 34 $ot :=< ob, [Pb] >$ *(Object Tree)*

Def. 35 $ob := instof(c), ob \in Ob, c \in C$ *(Object)*

Def. 36 $Pb := \{ot_1, ot_2, ..., ot_n\}, n \geq 0$ *(Object Properties - sub-trees of objects)*

Relations (R) connect two concepts (including predicates V), two objects, or an object with a concept and may have *probability distribution* Z (e.g., over time, over situations, over concepts' properties, etc.) (see Definition 38). A relation has an optional name, i.e., when the name is missing we have the implication relation. Probability distribution is provided to support *probabilistic reasoning*. By specifying relations with probability distributions we actually specify Bayesian Networks [Nea03] connecting the concepts and objects of an ontology. Note that KnowLang considers binary relations only, but there could be multiple relations relating the same concepts/objects.

Def. 37 $R := \{r_1, r_2, ..., r_n\}, n \geq 0$ *(Relations)*

Def. 38 $r :=< re_k, [rn], [Z], re_n >$ *(Relation, re - Relation Entity, Z - Probability Distribution)*
$re \in C \bigcup Ob \bigcup V$ *(C - Concepts, Ob - Objects, V - Predicates)*

Def. 39 $V := \{v_1, v_2, ..., v_n\}, n \geq 0$ *(Predicates)*

Def. 40 $v :=< C_v, S_v, be(O) >$ *(Predicate)*
$C_v \subset C, S_v \subset S$ *(C_v - Predicate's Concepts, S_v - Predicate's States)*

Predicates (V) are special KR structures that specify distinct inter-state relations or schemes for evaluation of complex states. For example, we can specify a predicate that verifies if the Motion System of a robot is operational. A predicate might be used by the KnowLang Reasoner to check whether an object (or the entire system) is in a specific state. Thus, a predicate (v) formally can be presented as tuple of predicate concepts (C_v), predicate states (S_v) and a Boolean expression over ontology ($be(O)$) that determines what conditions must hold to conclude that the predicate states are "active" (occupied) (see Definition 40.

KNOWLANG CONTEXTS

Def. 41 $Cx := \{cx_1, cx_2, ..., cx_n\}, n \geq 0$ *(Contexts)*

Def. 42 $cx :=< At, [Icx] >$ *(Context)*

Def. 43 $At := \{at_1, at_2, ..., at_n\}, n \geq 0$ *(Ambient Trees)*

Def. 44 $at :=< ct, Ca, [i] >$ *(Ambient Tree)*
$ct \in Ct$ *(Concept Tree hosted by an ontology)*
$Ca \subset C$ *(Ca - Ambient Concepts)*
$i \subset Icx$ *(i-Ambient Tree Interpretation)*

Def. 45 $Icx := \{i_1, i_2, ..., i_n\}, n \geq 0$ *(Context Interpretations)*

Contexts Cx are intended to extract the relevant knowledge from an ontology. Moreover, contexts carry interpretation for some of the meta-concepts (see Definition 42), which may lead to new interpretation of the descendant concepts (derived from a meta-concept - see Definition 8). We consider a very broad notion of context, e.g., the environment in a fraction of time or a generic situation such as currently-ongoing system action (e.g., observing or listening). Thus, a context must emphasize the key concepts in an ontology, which helps the inference mechanism narrow the domain knowledge (domain ontology) by exploring the concept trees down only to the emphasized key concepts.

Depending on the context, some low-level concepts might be subsumed by their upper-level parent concepts, just because the former are not relevant for that very context. For example, a robot wheel can be considered as a thing or as an important part of the robot's motion system. As a result, the context interpretation of knowledge will help the system deal with "clean" knowledge and the reasoning will be more efficient. A context (cx) consists of ambient trees (At) and optional context interpretations (Icx) (see Definition 42). An ambient tree (at) refers to a concept tree (ct) described by an ontology (o) and carries ambient concepts (Ca), part of the concept tree, and optional context interpretation (i).

The *ambient concepts* (see Definition 44) explicitly determine new level of deepness for their original concept tree, i.e., ambient concepts subsume all of their child concepts (if any). As result, when the system reasons about a particular context (expressed with ambient trees), the reasoning process does not consider those child concepts, but their ambient parents, which are far more generic, and thus less detailed. This technique reduces the size of the relevant knowledge, by temporarily removing from the concept trees all the ambient concepts' children (descendant concepts). We may think about ambient trees as filters the system applies at runtime to reduce the visibility of concepts of a concept tree. Note that this technique has been further developed in [VH12b].

KNOWLANG LOGICAL FRAMEWORK

Def. 46 $Lf :=< Fa, Rl, Ct >$ *(Logical Framework)*

Def. 47 $Fa := \{fa_1, fa_2, ..., fa_n\}, n \geq 0$ *(Facts)*

Def. 48 $fa := be(O) \rightarrow \boldsymbol{T}$ *(Fact - True statement over ontology)*

Def. 49 $Rl := \{rl_1, rl_2, ..., rl_n\}, n \geq 0$ *(Rules)*

Def. 50 $rl :=< be(O), do(A_{rl}) > | < be(O), do(V_{rl}) >$ *(Rule)*
$A_{rl} \subset A, V_{rl} \subset V$ *(A_{rl} - Rule's Actions, V_{rl} - Rule's Predicates)*

Def. 51 $Ct := \{ct_1, ct_2, ..., ct_n\}, n \geq 0$ *(Constraints)*

Def. 52 $ct := be(O)$ *(Constraint)*

The KnowLang Logical Framework helps developers realize the explicit representation of particular and general factual knowledge, in terms of additional rule-based predicates, names, connectives, quantifiers and identity. The Logical Framework (Lf) is composed of *facts* (Fa), *rules* (Rl) and *constraints* (Ct) (see Definition 46). Note that Lf's KR structures must be specified with ontology terms, i.e., predefined concepts, objects, predicates and relations. Facts define true statements in the ontologies (O) by applying Boolean expressions over ontology (see Definition 48). Rules relate hypotheses to conclusions where the former are expressed as Boolean expressions over ontology and the latter decide what actions to be performed or predicates to be enforced (see Definitions 50). A constraint is a Boolean expressions over ontology (see Definitions 52), e.g., constraints might negate the execution of particular actions or forbid the application of particular predicates. Constraints might be used to enforce knowledge consistency.

ASCENS KNOWLEDGE BASE OPERATORS

Def. 53 $Op :=< Ask, Tell, Oop >$ *(Knowledge Base Operators)*

Def. 54 $Ask := retrieve(Kc) \rightarrow Ip \lhd Kc$ *(query knowledge base)*

Def. 55 $Tell := update(Kc) \rightarrow Ip \rhd Kc$ *(update knowledge base)*

Def. 56 $Oop := fo(Oi) \rightarrow Ip \rhd Kc, Oi \subset O$ *(Inter-ontology Operators)*

ASCENS INFERENCE PRIMITIVES

Def. 57 $Ip := \{ip_1, ip_2, ..., ip_n\}, n \geq 0$ *(Inference Primitives)*

Def. 58 $ip := impl(FOL)|impl(FOPL)|impl(DL)$ *(Inference Primitive)*

The Knowledge Base Operators (Op) can be grouped into three groups: *ASK Operators* (retrieve knowledge from KBs), *TELL Operators* (update KB) and *Inter-Ontology Operators* (Oop) are intended to work on one or more ontologies (specified as a function $fo(Oi)$ over ontologies (Oi)) (see Definitions 53 through 56). The Inter-Ontology Operators are still under development, but overall they can be related to operations like *merging*, *mapping*, *alignment*, etc. Note that all the Knowledge Base Operators (Op) may imply the use of inference primitives (Ip).

The Inference Primitives (Ip) (see Definition 58) are algorithms for reasoning and knowledge inference needed by the KnowLang Reasoner. These primitives are implementation (denoted with $impl$ in Definition 58) of reasoning algorithms based on First Order Logic (FOL) [BL04] (and its extensions), First Order Probabilistic Logic (FOPL) [Hal90] and Description Logics (DL)[BN03]. FOPL increases the power of FOL by allowing us to assert in a natural way ”likely” features of objects and concepts via a probability distribution over the possibilities that we envision. Having logics with semantics gives us a notion of deductive entailment. Note that these algorithms together with the appropriate reasoning engines shall help the KnowLang Reasoner to query and update KB.

2.2 Meeting the Challenges

Both the KnowLang Specification Model and KnowLang Reasoner have been developed by taking into consideration some explicit challenges comprehensively described in our publications [VH12c, VHG12, VHG$^+$11].

2.2.1 Encoded versus Represented Knowledge

Developers may encode a large part of the ”a priori” knowledge (knowledge given to the system before the latter actually runs) in the implemented classes and routines. In such a case, the knowledge-represented pieces of knowledge (e.g., concepts, relations, rules, etc.) may complement the knowledge codified into implemented program classes and routines. For example, KnowLang actions could be based on classes and methods and a substantial concern about the KR of such actions is how to relate the knowledge expressed with actions to implemented methods and functions. A possible solution is to map KR concepts and objects to program classes and objects respectively.

To properly represent the program implementation (classes, methods, etc.) in the KB, all the concepts and objects have an *IMPL Property* that relates a KnowLang structure to its program counterpart, if any. For example, a KnowLang concept might be specified with an IMPL property to link the concept to a program class or method. The following is the grammar definition supporting that [Vas12c].

```
Concept-Impl := IMPL { Impl-Reference }
```

2.2.2 States, Situations, Goals and Policies

A big challenge is *"how to express situations and reason about the same"*. Situations trigger self-adaptive behavior (see Section 2.4.2) and it is very important to allow the reasoner to recognize them. To support this approach, KnowLang has introduced the *STATE explicit concepts* (see Definiton 23 in Section 2.1). This helps each KnowLang concept to be specified with a set of important states the concept instances can be in. Thus, we explicitly specify a variety of states for important concepts (e.g., states "operational" and "non-operational" for the robot's Motion System). A KnowLang state is specified as a Boolean expression over ontology where we can use activation of events, execution of actions or changes in properties to build a state's Boolean expression [Vas12c]. Further, to facilitate the evaluation of complex states, we specify *PREDICATES* (see Definition 40 in Section 2.1). Complex states (e.g., system states) are the product of other states (e.g., the states of the system's components). States (usually system states) are also used to specify *GOALS*, another class of KnowLang explicit concepts (see Definition 22 in Section 2.1). Goals participate in the specification of KnowLang policies. A goal can be specified as a transition from a state to another. Recall that policies and situations participate in KnowLang relations (see Definition 19 in Section 2.1) that drive the *self-adaptive behavior* (see Section 2.4.2). Therefore, because every situation is explicitly related to a state (a situation is determined by a state), it is relatively easy to check for the feasibility of a policy triggered by a specific situation, i.e., the policy's goal must have the same departing state as the situation's state.

2.2.3 Converting Sensory Data to KR Symbols

One of the biggest challenges is *"how to map sensory raw data to KR symbols"*. Our approach to this problem is to specify special explicit concepts called *METRICS* (see Definition 17 in Section 2.1). In general, a SCE system has sensors that connect it to the world and eventually help it to listen to its internal components. These sensors generate raw data that represent the physical characteristics of the world. The problem is that these low-level data streams must be: 1) converted to programming variables or more complex data structures that represent collections of sensory data; 2) those programing data structures must be labeled with KR Symbols. Hence, it is required to relate encoded data structures with KR concepts and objects used for reasoning purposes. In our approach, we assume that each sensor is controlled by a software driver (e.g., specified in SCEL and implemented in Java) where appropriate methods are used to control the sensor and read data from it. Both the *sensory data* and *sensors* should be represented in the KB by using *METRIC* explicit concepts and instantiate objects of these concepts. By specifying a METRIC concept we introduce a *class of sensors* to the KB and by specifying objects, instances of that class, we give the actual KR of a real sensor. KnowLang allows the specification of four different types of metrics [Vas12c]:

- RESOURCE - measure SC resources like capacity;
- QUALITY - measure SC qualities like performance, response time, etc.;
- ENVIRONMENT - measure environment qualities and resources;
- ENSEMBLE - measure SCE qualities and resource; might be a function of multiple SC metrics both of RESOURCE and QUALITY type.

2.3 KnowLang Syntax

We used the Backus-Naur Form (BNF) notation [Knu64] to describe the syntax of the language and formally specify the KnowLang Grammar [Vas12c]. This helps the KnowLang framework to process sentences written in the KnowLang language. BNF [Knu64] is a powerful meta-language that allows a context-free grammar specification. A partial presentation of the KnowLang Grammar in BNF is the following [Vas12c]:

```
KL-Spec := bof Knowledge-Spec eof
Knowledge-Spec := Spec-References KL-Spec-Units
Knowledge-Spec := KL-Spec-Units
KL-Spec-Units := KL-Corpuses KL-Operators Inference-Primitives
....
KL-Spec-Units := KL-Corpuses
KL-Spec-Units := KL-Operators
KL-Spec-Units := Inference-Primitives
```

As shown, the full KnowLang context-free grammar specification is obtained by the reduction of the (*KL-Spec -> bof Knowledge-Spec eof*) rule, which determines that a KB specified with KnowLang consists of *specification units*, each formed by a combination of *knowledge corpuses*, *KB operators* and *inference primitives*. Due to the complex structure of the KnowLang specification model (see Section 2.1) where each tier has its own structure, the complete KnowLang Grammar's specification cannot be presented here (please refer to [Vas12c] for the full KnowLang Grammar in BNF). Instead, we present an abstraction of the KnowLang Grammar, i.e., a meta-grammar. The following is a generic meta-grammar in Extended BNF [Knu64] presenting the syntax rules for specifying KnowLang tiers.

```
GroupTier := FINAL?  GroupTierId  {  Tier+  }
Tier := FINAL?  TierId TierName?  {  TierClause+  }
TierClause := FINAL?  ClauseId  ClauseName? {  Data*  }
Data := PredefType | ConceptNames | BlnExpr | Reference | Number
ConceptNames := ConceptName [,ConceptName]*
```

As shown, in general a KnowLang tier is syntactically specified with a *tier identifier* (predefined KnowLang name), an optional *name* and a *content block* bordered by curly braces. Moreover, we distinguish two syntactical tier types: *single tiers* ($Tier$) and *group tiers* ($GroupTier$) where the latter comprise a set of single tiers. Each single tier has an optional *name* ($TierName$) and comprises a set of *tier clauses* ($TierClause$), which are composed of a *clause identifier*, an optional *clause name* and optional *data* ($Data$). The latter presents a predefined KnowLang type (e.g., $METRIC$ type), a collection of names (e.g., concept names or objects names), a Boolean expression over ontology, an implementation reference (e.g., $IMPL\{Sensors.LightSensor.getSourceAngle()\}$) or a number. Note that identifiers participating in KnowLang expressions are either simple, consisting of a single identifier, or qualified, consisting of a sequence of identifiers separated by "." tokens. Identifiers could be concept names, object names, relation names, predicate names, property names or function names, and it is important to specify them with their qualified name, e.g., pointing where a concept resides in a concept tree. When we use ".." token, we let the KnowLang Reasoner find the specified identifier presuming it is unique in the current tree.

2.4 KnowLang Reasoner

A very challenging task in WP3 is the R&D of the inference mechanism providing for knowledge reasoning and awareness. As described in the first deliverable of WP3 [VHG$^+$11], the initial strategy was to use high-order inference engines based on FOL and DLs and driven by the inference primitives defined by KnowLang (see Definition 58 in Section 2.1). However, in order to support reasoning about self-adaptive behavior and to provide a KR gateway for the ASK and TELL Operators (see Definitions 53 through 56 in Section 2.1), in this second year of the project, we started working on a distinct KnowLang Reasoner. The reasoner communicates with the system via ASK and TELL Operators (forming a communication interface) and operates in the KR Context, a context formed by the represented knowledge (See Figure 2).

The KnowLang Reasoner will be supplied as a component hosted by the ASCENS system and thus, it will run in the system's Operational Context as any other system's component. However, it

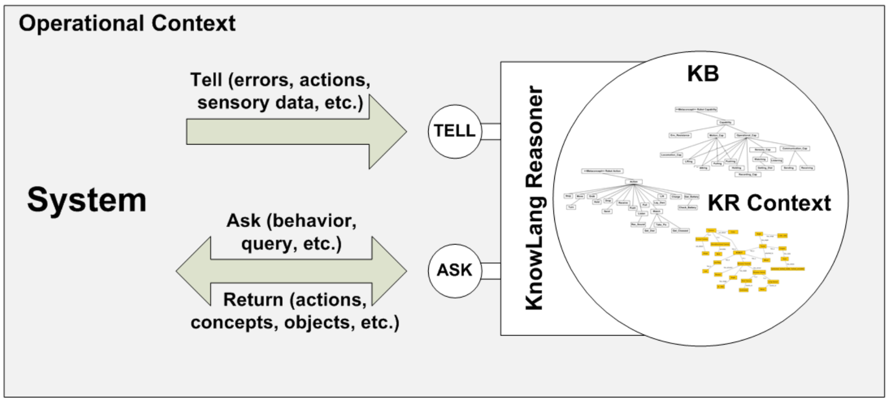

Operational Context
System
Tell (errors, actions, sensory data, etc.)
Ask (behavior, query, etc.)
Return (actions, concepts, objects, etc.)
TELL
ASK
KnowLang Reasoner
KB
KR Context

Figure 2: KnowLang Reasoner

operates in the KR Context and on the KR symbols (represented knowledge). The system talks to the reasoner via the ASK and TELL Operators allowing for knowledge queries and knowledge updates (See Figure 2). Upon demand, the KnowLang Reasoner can also build up and return a self-adaptive behavior model consisting of a chain of actions to be realized in the environment or within the system.

2.4.1 ASK and TELL Operators

In this second year of the project, we started working on a predefined set of *ASK* and *TELL Operators* for KnowLang. TELL Operators feed the KR Context with important information driven by errors, executed actions, new sensory data, etc., thus helping the KnowLang Reasoner to 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. The following presents generic algorithms of how both classes of KB Operators operate with knowledge.

TELL Algorithm

1. The system tells the KnowLang Reasoner about errors, sensory data, execution of actions or actual updates.

2. The KnowLang Reasoner switches to the KR Context and maps the input to KR symbols.

3. The KnowLang Reasoner updates the KB, e.g., updates concepts / objects or adds new concepts / objects and changes states.

ASK Algorithm

1. The system asks for a self-adaptive behavior, rule-based behavior, current state or current situation.

2. The KnowLang Reasoner switches to the KR Context and maps the input to KR symbols.

3. The KnowLang Reasoner processes the query to get behavior actions or retrieve information and eventually updates the KB.

4. The KnowLang Reasoner builds the output and returns the result to the system.

So far, we have developed the operational semantics of the following TELL and ASK Operators [Vas12d]:

- $TELL\_ERR$ - 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.

The rest of this section is a brief presentation of the operational semantics of two KB Operators.

Operational Semantics of the $TELL\_SENSOR$ Operator.

$TELL\_SENSOR$ Operator is used by the system to tell the KnowLang Reasoner about new sensory data, i.e., data obtained by one of the system's sensors, e.g., light sensor, microphone, etc. In order to update the KB with the recent *sensory data*, the system passes it through the $TELL\_SENSOR$ Operator along with the *data source*, i.e., the program object, class and/or method implementing that sensor. The following rules reveal the operational semantics of the $TELL\_SENSOR$ Operator. Note that in the definitions below, σ states for the system *Operational Context* (OC) and σ' states for the system *KR Context* (KRC). Moreover, for clarity reasons (to show that the system stays in KRC while the KnowLang Reasoner is operating within it), we do not show the change in KRC after updates have been made in that context.

$$(1)\,\frac{\sigma \xrightarrow{tell\_sensor(d,s)} \sigma'}{\langle TELL\_SENSOR(d,s),\sigma'\rangle \longrightarrow \langle findMetricConcept(s),\sigma'\rangle}\ \textit{d-data, s-source}$$

$$(2)\,\frac{\sigma \xrightarrow{tell\_sensor(d,s)} \sigma' \langle findMetricConcept(s),\sigma'\rangle \longrightarrow \langle c,\sigma'\rangle}{\langle findMetricObject(c,s),\sigma'\rangle \longrightarrow \langle o_m,\sigma'\rangle}\ \textit{c-metric concept, } o_m\textit{-metric object}$$

$$(3)\ \frac{\sigma \xrightarrow{tell\_sensor(d,s)} \sigma' \langle findMetricObject(c,s),\sigma'\rangle \longrightarrow \langle o_m,\sigma'\rangle}{\langle update(o_m,d),\sigma'\rangle \longrightarrow \langle o'_m,\sigma'\rangle}\ o'_m\textit{-updated metric object}$$

$$(4)\ \frac{\langle update(o_m,d),\sigma'\rangle \longrightarrow \langle o'_m,\sigma'\rangle \langle findMetricEvents(o'_m),\sigma'\rangle \longrightarrow \langle E_m,\sigma'\rangle}{\forall e_m \in E_m \bullet \langle fireEvent(e_m),\sigma'\rangle \longrightarrow \langle o_e,\sigma'\rangle}\ o_e\textit{-event instance (fired event)}$$

$$(5)\ \frac{\langle fireEvent(e),\sigma'\rangle \longrightarrow \langle o_e,\sigma'\rangle \langle findDependedObjects(e),\sigma'\rangle \longrightarrow \langle O_d,\sigma'\rangle}{\forall o_d \in O_d \bullet \langle setCurrentState(o_d,e),\sigma'\rangle \longrightarrow \langle o_d.STATE,\sigma'\rangle}$$

$$(6)\ \frac{\langle fireEvent(e),\sigma'\rangle \longrightarrow \langle o_e,\sigma'\rangle \forall o_d \in O_d \bullet \langle setCurrentState(o_d,e),\sigma'\rangle \longrightarrow \langle o_d.STATE,\sigma'\rangle}{\langle findCurrentSituation(),\sigma'\rangle \longrightarrow \langle si,\sigma'\rangle}$$

$$(7)\ \frac{\langle fireEvent(e),\sigma'\rangle \longrightarrow \langle o_e,\sigma'\rangle \langle findCurrentSituation(),\sigma'\rangle \longrightarrow \langle si,\sigma'\rangle}{\langle recordEventHistory(o_e,si),\sigma'\rangle \longrightarrow \left\langle \overleftarrow{e_{si}},\sigma'\right\rangle}$$

As shown in Rule 1, the call of the $tell\_sensor()$ function (a method implementing the system call of the $TELL\_SENSOR$ Operator) triggers a context switching $\sigma \xrightarrow{tell\_sensor(d,s)} \sigma'$, i.e., the process control is passed to the KnowLang Reasoner, which operates in the KRC only. Further, this context switching initiates an internal for KRC call of the $TELL\_SENSOR$ Operator, which triggers the retrieval of the *metric concept* specified in the KB to represent the sensor's class implemented in the program. The $findMetricConcept(s)$ function is used to denote the execution of a traversal algorithm that finds a metric concept by an *IMPL reference* string (s carries information about the sensor implementation, e.g., a class). Then, if the metric concept has been successfully found, the reasoner looks up the *concept instance* representing the sensor's object in the program implementation (denoted in Rule 2 with the $findMetricObject(c, s)$ function). If the concept instance is successfully found, then the reasoner updates that instance accordingly (denoted in Rule 3 with the $update(o_m, d)$ function). Next, the reasoner looks up and fires all the events specified to be activated by a change in this specific metric (see Rule 4 and the abstract functions $findMetricEvents(o'_m)$ and $fireEvent(e_m)$ respectively). Note that KnowLang events can be specified to be activated by a data change in specific metrics. The following fragment of the KnowLang Grammar [Vas12c] demonstrates that:

```
Event-Activ-Fact := CHANGED { Metric-Name }
```

When an event is fired, actually the reasoner creates a new event instance (event object) in the KRC. Further, the reasoner looks up all the concept instances whose states depend on the existence of that event (see Rule 5 and abstract function $findDependedObjects(e)$). Recall that states in KnowLang are expressed as Boolean expression over ontology (see Section 2.2.2). The occurrence of an event can be used within such expressions and thus, events can be used to specify states. The following are fragments of the KnowLang Grammar [Vas12c] demonstrating that:

```
State-Body := Bln-Expr

Bln-Reln := OCCURRED ( Event-Name )
```

Therefore, in order to keep the KB consistent, every time when an event is fired in the KRC, the KnowLang Reasoner finds the objects (concept instances) whose states depend on that event and sets their states accordingly (denoted in Rule 5 with $setCurrentState(o_d, e)$). As shown in Rule 6, once all the objects have been updated accordingly, the reasoner looks up the new situation in regards to the global state change (denoted in Rule 6 with $findCurrentSituation()$). Finally, the fired event is recorded in the *event history* of the current situation (see Definition 25 in Section 2.1).

Operational Semantics of $ASK\_BEHAVIOR$ Operator.

ASK_BEHAVIOR Operator is used by the system to ask the KnowLang Reasoner for self-adaptive behavior considering the current situation the system is in. The following rules reveal the operational semantics of the ASK_BEHAVIOR Operator. Again, σ states for OC and σ' states for KRC. Also, again for clarity reasons, we do not show the change in KRC after updates have been made in that context.

$$(8)\,\frac{\sigma \xrightarrow{ask\_behavior()} \sigma'}{\langle ASK\_BEHAVIOR, \sigma' \rangle \longrightarrow \langle findCurrentSituation(), \sigma' \rangle}$$

$$(9)\,\frac{\sigma \xrightarrow{ask\_behavior()} \sigma' \langle findCurrentSituation(), \sigma' \rangle \longrightarrow \langle si, \sigma' \rangle}{\langle findSitnPolcyRltns(si), \sigma' \rangle \longrightarrow \langle R_{si}, \sigma' \rangle}$$

$$(10)\,\frac{\sigma \xrightarrow{ask\_behavior()} \sigma' \langle findSitnPolcyRltns(si), \sigma' \rangle \longrightarrow \langle R_{si}, \sigma' \rangle}{\langle max(R_{si}), \sigma' \rangle \longrightarrow \langle \pi_{si}, \sigma' \rangle}\ \textit{probability distribution determines } \pi_{si}$$

$$(11)\ \langle \pi, \sigma' \rangle \longrightarrow \langle applyPolicy(\pi), \sigma' \rangle\ \textit{evaluate policy } \pi \textit{ in context } \sigma'$$

$$(12)\,\frac{\langle \pi_{si}, \sigma' \rangle \longrightarrow \langle applyPolicy(\pi_{si}), \sigma' \rangle \forall n_\pi \in N_\pi \bullet \langle n_\pi, \sigma' \rangle \longrightarrow \langle TRUE, \sigma' \rangle}{\langle map(\pi_{si}, N_\pi, A_\pi, Z), \sigma' \rangle \longrightarrow \langle < A'_\pi, Z' >, \sigma' \rangle}\ A'_\pi \subseteq A_\pi$$

$$(13)\,\frac{\langle \pi_{si}, \sigma' \rangle \longrightarrow \langle applyPolicy(\pi_{si}), \sigma' \rangle \langle map(\pi_{si}, N_\pi, A_\pi, Z), \sigma' \rangle \longrightarrow \langle < A'_\pi, Z' >, \sigma' \rangle \langle max(Z'), \sigma' \rangle \longrightarrow \langle z, \sigma' \rangle}{\langle getProbableActions(< A'_\pi, Z' >, z), \sigma' \rangle \longrightarrow \langle < A''_\pi, z >, \sigma' \rangle}$$

$$(14)\,\frac{\begin{array}{c}\langle \pi_{si}, \sigma' \rangle \longrightarrow \langle applyPolicy(\pi_{si}), \sigma' \rangle \langle map(\pi_{si}, N_\pi, A_\pi, Z), \sigma' \rangle \longrightarrow \langle < A'_\pi, Z' >, \sigma' \rangle \\ \langle getProbableActions(< A'_\pi, Z' >, z), \sigma' \rangle \longrightarrow \langle < A''_\pi, z >, \sigma' \rangle\end{array}}{\langle recordBehavior(\pi_{si}, A''_\pi), \sigma' \rangle \longrightarrow \left\langle b^\pi_{si}, \sigma' \right\rangle}\ A''_\pi \subseteq A_{si}$$

$$(15)\,\frac{\sigma \xrightarrow{ask\_behavior()} \sigma' \langle recordBehavior(\pi_{si}, A_\pi), \sigma' \rangle \longrightarrow \left\langle b^\pi_{si}, \sigma' \right\rangle}{\sigma' \xrightarrow{return(b^\pi_{si})} \sigma}\ A_\pi \subseteq A_{si}$$

As shown in Rule 8, to ask for behavior, the system calls the $ask\_behavior()$ function (a method implementing the system call of the ASK_BEHAVIOR Operator), which triggers a context switching $\sigma \xrightarrow{ask\_behavior()} \sigma'$. This passes the process control to the KnowLang Reasoner, which operates in the KRC only. Further, this context switching initiates an internal for KRC call of the ASK_BEHAVIOR Operator, which starts an internal operation (denoted with the $findCurrentSituation()$ abstract function) to find the current situation the system is currently in. The current situation will be approximately determined based on the *global system state*. Once the current situation is successfully determined (see the second premise in Rule 9), the reasoner needs to find all the policies associated with that situation. Thus, the reasoner looks up all the *situation-policy relations* the current situation participates in (denoted with the $findSitnPolcyRltns(si)$ - see the conclusion in Rule 9). Next, the relation with the *highest probability rate* is selected (recall that KnowLang Relations may be associated with a probability rate - see Definition 38 in Section 2.1), which helps to determine the *most appropriate policy* for that particular situation (see the conclusion in Rule 10). The evaluation of the selected policy actually triggers its application (see Rule 11). The evaluation of a KnowLang policy triggers a *mapping operation* where any *policy condition* that is held (the conditions are Boolean expressions) is mapped to appropriate actions with eventual *probability rate* (see Definition 19 in Section 2.1). This operation selects *pairs "actions subset"-"probability rate"* (see the conclusion in Rule 12). Next, the reasoner selects from these pairs the one with the highest probability rate to extract the *subset of actions* to be executed (see the last premise and conclusion in Rule 13). The extracted subset of possible actions has to be recorded as a *behavior model* (see the conclusion in Rule 14 where this is denoted with the $recordBehavior(\pi_{si}, A''_\pi)$ abstract function). Finally, the KnowLang Reasoner returns the recorded behavior model to the system, which causes a context switching back to OC (see Rule 15). Note that the behavior model must comprise only actions allowed to be executed from the actual situation (see Definition 25 in Section 2.1).

2.4.2 KR for Self-adaptive Behavior with KnowLang

KnowLang has intrinsic features supporting KR for autonomic systems (recall that ASCENS is a class of atomic systems). An *autonomic system* [KC03, VH10] is considered to be a self-adaptive system that changes its behavior in response to stimuli from its execution and operational environment. Such behavior is considered *autonomic* and *self-adaptive* [VH10] and is intended to drive a system in situations requiring adaptation. Any long-running system is subject to uncertainty in its execution environment due to potential changes in requirements, business conditions, available technology, etc. Thus, it is important to capture and cater for uncertainty as part of the development process. Failure to do so may result in systems that are too rigid to be fit for purpose, which is of particular concern for the domains that typically make use of self-adaptive technology, e.g., ASCENS. We hypothesize that modeling uncertainty and developing mechanisms for managing it as part of KR&R will lead to systems that are:

- more expressive of the real world;
- fault tolerant due to fluctuations in requirements and conditions being anticipated;
- flexible and able to manage dynamic changes.

The ability to represent knowledge providing for *self-adaptive behavior* is an important factor in dealing with uncertainty. In our approach, the autonomic self-adaptive behavior is provided by *policies, events, actions, situations*, and *relations* between policies and situations (see Definitions 18 through 25 in Section 2.1).

Ideally, policies are specified to handle specific situations, which may trigger the application of policies. A policy exhibits a behavior via actions generated in the environment or in the system itself. Specific conditions determine, which specific actions (among the actions associated with that policy - see Definition 19 in Section 2.1) shall be executed. These conditions are often generic and may differ from the situations triggering the policy. Thus, the behavior not only depends on the specific situations a policy is specified to handle, but also depends on additional conditions. Such conditions might be organized in a way allowing for synchronization of different situations on the same policy. When a policy is applied, it checks what particular conditions are met and performs the associated actions via special mappings (see $map(N_\pi, A_\pi, [Z])$ in Definition 19 in Section 2.1). An optional probability distribution (Z) may additionally restrict the action execution. Although initially specified, the probability distribution at the mappings is recomputed after the execution of any involved action. The re-computation is based on the consequences of the action execution, which allows for reinforcement leaning.

The cardinality of the *policy-situation relationship* is many-to-many, i.e., a situation might be associated with many policies and vice versa. The set of *policy situations* (situations triggering a policy) is open-ended, i.e., new situations might be added or old might be removed from there by the system itself. Moreover, with a set of *policy-situation relations* we may grant the system with an initial *probabilistic belief* (see Definition 19) that certain situations require specific policies to be applied. Runtime factors may change this probabilistic belief with time, so the most likely situations a policy is associated with can be changed. For example, the successful rate of actions execution associated with a specific situation and a policy may change such a probabilistic belief and place a specific policy higher in the ”list” of associated policies, which will change the behavior of the system when a specific situation is to be handled. Note that situations are associated with a state (see Definition 25) and a policy has a goal (see Definition 19), which is considered as a transition from one state to another (see Definition 2). Hence, the *policy-situation relations* and the employed *probabilistic beliefs* may help a cognitive system what desired state to choose, based on past experience.

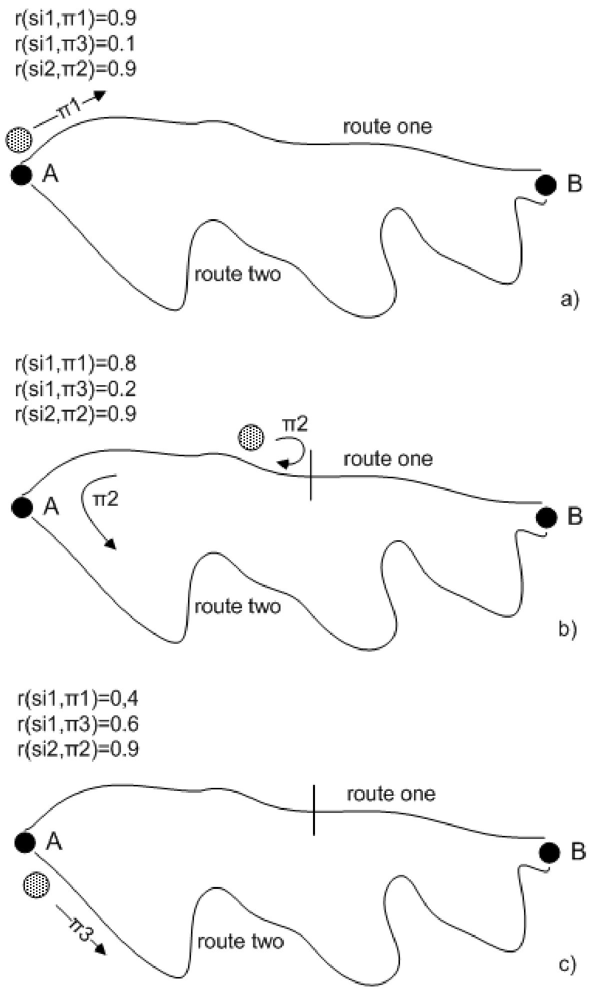


Figure 3: A marXbot Self-adaptation Case Study

As a proof of concept, we applied the approach to one of the ASCENS case studies - the Ensemble of Robots Case Study. To illustrate autonomic behavior based on this approach, let us suppose that we have a marXbot robot that carries items from point A to point B by using two possible routes - route one and route two (see Figure 3).

A situation si_1:*"robot is in point A and loaded with items"* will trigger a policy π_1:*"go to point B via route one"* if the relation $r(si_1, \pi_1)$ has the higher probabilistic belief rate (let's assume that such a rate has been initially given to this relation because *route one* is shorter - see Figure 3.a). Any time when the robot gets into situation si_1 it will continue applying the π_1 policy until it gets into a situation si_2:*"route one is blocked"* while applying that policy. The si_2 situation will trigger a policy π_2:*"go back to si_1 and then apply policy π_3"* (see Figure 3.b). Policy π_3 is defined as π_3:*"go to point B via route two"*. The unsuccessful application of policy π_1 will decrease the probabilistic belief rate of relation $r(si_1, \pi_1)$ and the eventual successful application of policy π_3 will increase the probabilistic belief rate of relation $r(si_1, \pi_3)$ (see Figure 3.b). Thus, if route one continues to be blocked in the future, the relation $r(si_1, \pi_3)$ will get to have a higher probabilistic belief rate than the relation $r(si_1, \pi_1)$ and the robot will change its behavior by choosing route two as a primary route (see Figure 3.c). Similarly, this situation can change in response to external stimuli, e.g., *route two* got blocked or a *"route one is obstacle-free"* message is received by the robot.

Another self-adaptation case study is presented in [WHM$^+$12] where the experience of action execution changes the probability distribution within the currently active policy, i.e., at the level of policy mappings.

2.5 KnowLang Toolset

To support the R&D of KnowLang, we started working on the *KnowLang Toolset*. Originally, the toolset is intended to provide a suitable development environment for KR where we can write KR specifications in the KnowLang notation by using visual modeling tools and check for the syntactical integrity and consistency of the KR models. As designed, the KnowLang Toolset consists of the following components:

- Visual Editor;
- Grammar Compiler;
- KnowLang Parser;
- Consistency Checker;
- Semantics Analyzer.

Components from these tools are to be linked together to form a special *KnowLang Compiler* that shall compile the KR models specified in KnowLang into *KnowLang Binary* (see Figure 4). The KnowLang Binary is the core of the KB (Knowledge Base), which is operated by the KnowLang Reasoner. Note that the KnowLang Reasoner is a distinct KnowLang component intended to be integrated within the systems using KnowLang KR, but it also can be integrated in the KnowLang Toolset where it can be used for testing and behavior analysis.

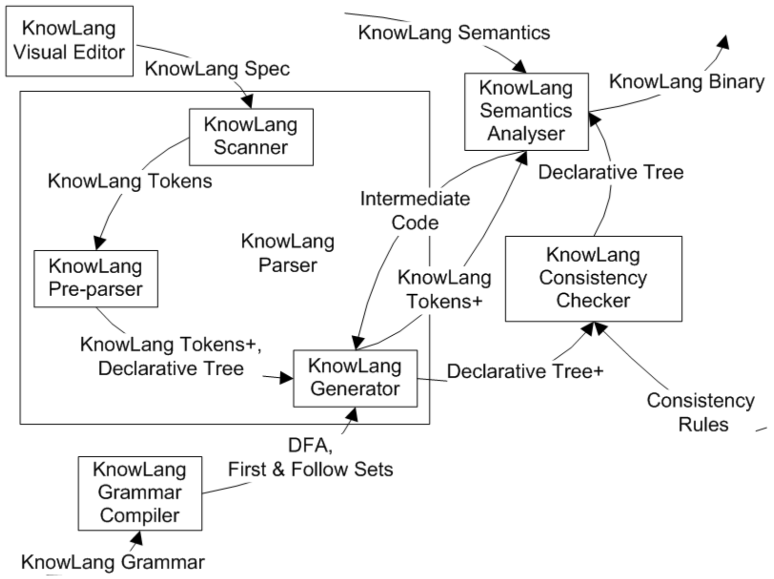


Figure 4: KnowLang Compiler

Currently, we have developed the first version of both Visual Editor and Grammar Compiler. This allows us to have limited but sufficient capabilities for writing simple KR models with KnowLang.

The KnowLang Toolset provides a distinct set of software components part of the ASCENS Software Component Repository presented in D8.2 [HBGK12].

3 KR Models for ASCENS Case Studies

To specify knowledge with KnowLang you need to think about 1) domain concepts and their properties and functionalities (e.g., actions that can be realized in the environment); 2) important states of major concepts; 3) objects as realization of concepts; 4) relations to show how concepts and objects are related to each other; 5) self-adapting scenarios for the system in question, e.g., eventual problematic situations with desired outcome; 6) remarkable behavior in terms of policies driving the system out of specific situations; 7) other important specifics that can be classified as concepts (could be explicit) and objects, e.g., SLO (service-level objectives), QoS proiperties, system sensors, group formations, etc.

In this second year of the project, we implemented initial KR models for all the three ASCENS case studies [Vas12b]. Although, still these models need to be both refined and completed, we present some parts of the KR Model for the marXbot case study [VH12d].

Figure 5 depicts the graphical representation of a concept tree specified with KnowLang. As shown, the three has a tree root *Thing*. The concept *Thing* is determined by the meta-concept *Robot Thing*, which carries information about the interpretation of the root concept *Thing* such as *"Thing is anything that can be related to the robot"*. According to this concept tree there are two categories of

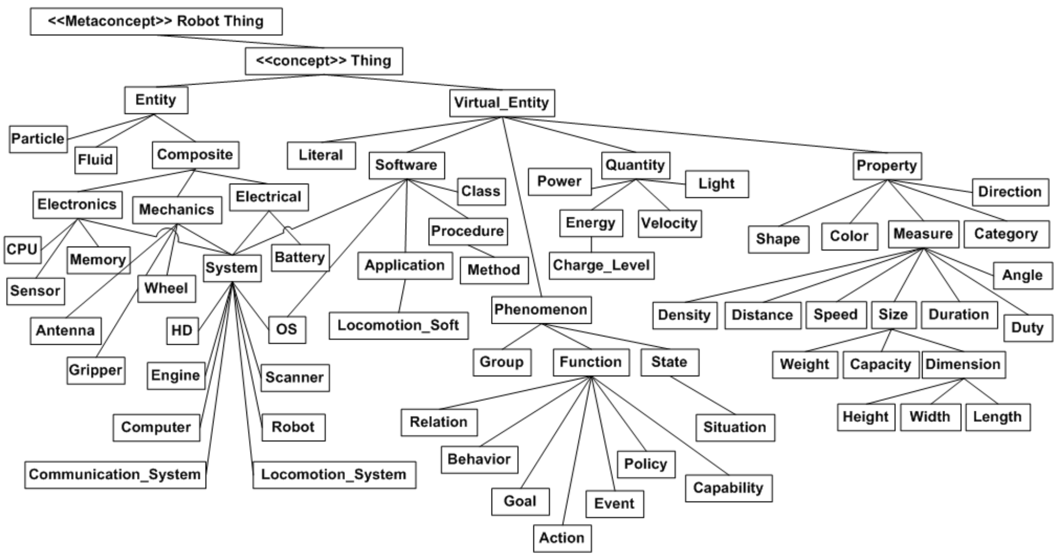


Figure 5: Concept Tree: "Robot Thing"

things in a robot: *entities* (physical entities) and *virtual entities* where both are used to organize the vocabulary in the internal robot domain. Note that the explicit concepts (see Figure 1 in Section 2.1) are presented as concepts in this concept tree - qualified path *Thing->Virtual_Entity->Phenomenon*, i.e., in this ontology tree, the explicit concepts inherit the concepts *Phenomenon*, *Function* and *State*. The following KnowLang code presents the actual specification of the *Locomotion_System* concept.

```
CONCEPT Locomotion_System {
  CHILDREN {}
  PARENTS { SC.Thing..System }
  STATES { STATE operational {} STATE on {} STATE off {} }
  PROPS {
    PROP engine { TYPE {SC.Thing..Engine} CARDINALITY {1} }
    PROP wheel { TYPE {SC.Thing..Wheel} CARDINALITY {5} }
    PROP locomotion_soft { TYPE {SC.Thing..Locomotion_Soft}  CARDINALITY {1} }
    PROP battery { TYPE {SC.Thing..Battery}  CARDINALITY {1} }
  }
  FUNCS {
   FUNC move { TYPE {SC.Action.Move } }
   FUNC stop { TYPE {SC.Action.Stop } }
   FUNC turn { TYPE {SC.Action.Turn } }
  }
  IMPL { Robot.LocomotionSystem }
}
```

Recall that KnowLang concepts have a *STATES* attribute that may be associated with a set of possible state values the concept instances may be in. The KnowLang states are intrinsic concepts descending from the *State* concept (see Figure 5). In general, a system may occupy a new state when values of concept properties have been changed or some events or actions have occurred in the system or the environment. Therefore, a state can be determined by values held by concept properties, raised events or executed actions. To represent a state, we construct a Boolean expression over ontology. A state of a complex concept might be the product of the states of its properties. Only significant states should be specified and evaluated by using predicates. For example, the predicate *Is_Operational* evaluates whether a concept instance is in operational state:

```
PREDICATE Is_Operational {
  ENTRIES {SC.Thing..Locomotion_System}
  STATES {Operational} EXPRESSION {...}
}
```

Further, we specify the concept *Capability* (see Figure 6), which descends from the *Function* concept (see Figure 5) and couples a function with elements that increase its depth, scope, productivity, etc. Capability may carry information about possible range, limits, etc.. The concept *Action* (see Figure 7) descends from the *Function* concept (see Figure 5) and defines the entire robot's functionality as possible actions. Moreover, actions are used to specify the functions (functionality) of a concept. In

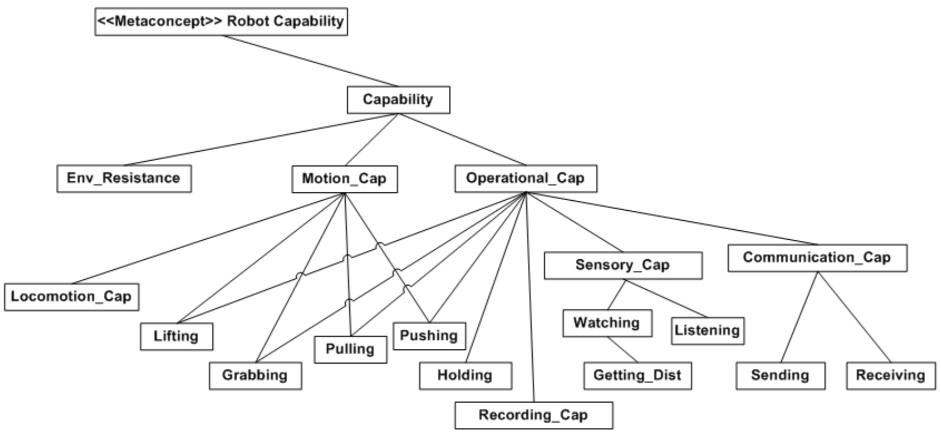


Figure 6: Concept Tree: "Robot Capability"

addition to the concept trees, to build the ontology concept maps, we also specify *relations*. In general, we specify relations connecting concepts and objects in the Robot SC Ontology. For example, possible relations are the following:

```
RELATION Instance_Of {
  RELATION_PAIR {object.robot[1].locomotion_system, Thing..Locomotion_System} }
RELATION Part_Of { RELATION_PAIR {object.locomotion_system, object.robot[1]} }
RELATION Engrouped { RELATION_PAIR {object.robot[1], object.robot[2]} PROBABILITY {1} }
```

Figure 8 depicts a *concept map* built over specified relations for the Robot SC Ontology.

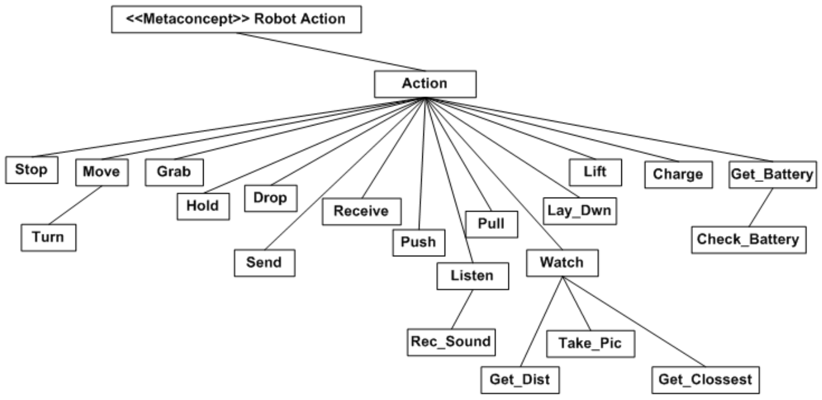


Figure 7: Concept Tree: "Robot Action"

To complete the KB we also need to specify *object trees* presenting the important concept instances in the domain of interest, e.g., the content of the robot in terms of components. Note that the object trees realize the concepts, i.e., they inherit the properties and functions of the concepts they instantiate. For example, Figure 9 depicts the specification of the *Gripper* concept and its realization, the object *gripper_23*.

We specify *facts* to impose true statements in ontology, e.g., *implication*. The following examples present some facts:

```
FACT {
   Predicate.Work_With(object.robot[1], object.robot[2]) =>
      Predicate.Engrouped(object.robot[1], object.robot[2]) }
FACT {
   Predicate.Is_Operational(THIS.locomotion_system) AND
      Predicate.Obstacle_Free(THIS) => Predicate.Can_Move(THIS) }
```

We also specify *rules* to impose simple behavior, e.g.:

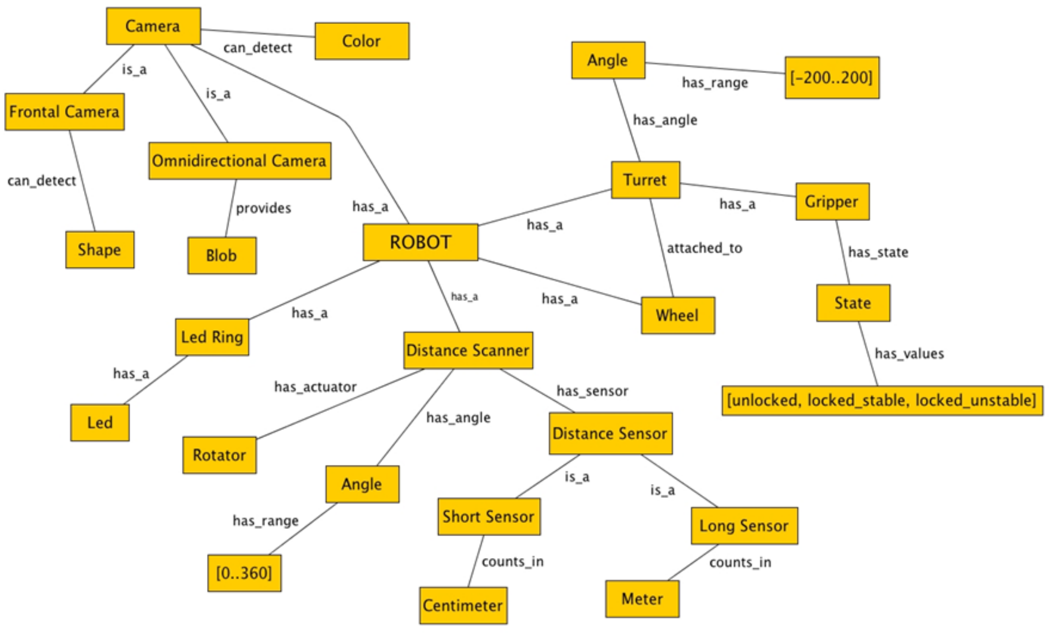


Figure 8: Concept Map: "Robot Relations"

```
RULE {
    IF  NOT Predicate.Can_Move(THIS) THEN  { DO {Action.Check_Battery(THIS..battery);} } }
RULE {
    IF  NOT Predicate.Can_Move(THIS) AND Action.Get_Battery(THIS..battery) > 0.5
    THEN {
        DO {Action.Get_Dist(THIS, Action.Get_Closest(THIS, ENV.Thing..Obstacle)); } } }
```

Rules, also can be used to imply predicates, e.g.:

```
RULE {
    IF Action.Get_Battery(THIS..battery) > 0.9  THEN  {
        Predicate.Charged(THIS..battery)
    } ELSE {
        NOT Predicate.Charged(THIS..battery)
    }
}
```

In additon, *constraints* may be used to constraint the behavior of the system or impose predicates, e.g.:

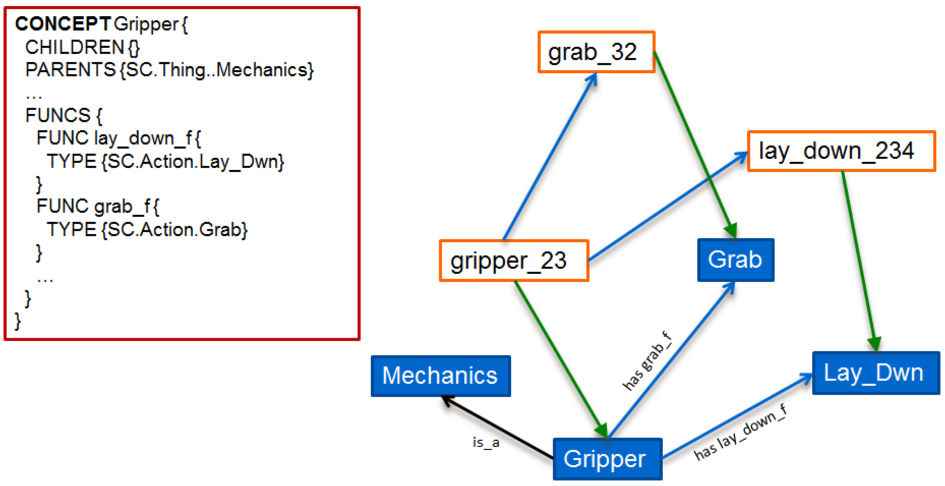
```
CONSTRAINT {
   IF Action.Get_Battery(THIS..battery) < 0.1 THEN { NOT Action.Move(THIS) }

CONSTRAINT {
   IF Predicate.Is_Operational(THIS.locomotion_system) THEN  {
       Action.Get_Battery(THIS..battery) > 0.5  AND
       Predicate.Is_Operational (THIS..wheel[1])  AND
       Predicate.Is_Operational (THIS..wheel[2])  AND
       Predicate.Is_Operational (THIS..wheel[3])  AND
       Predicate.Is_Operational (THIS..wheel[4])  AND
       Predicate.Is_Operational (THIS..wheel[5])  AND
       Predicate.Is_Operational (THIS..engine)  AND
       Predicate.Is_Operational (THIS..locomotion_soft)  AND
       Predicate.Is_Running (THIS..locomotion_soft)
    }
}
```

Constraints can also be used to impose data restrictions, e.g., let's presume we want two robots to have different first goals:

```
CONSTRAINT { robot[1].goal[1] <> robot[2].goal[1]; }
```

Finally, we need to specify important situations and policies driving the system in those situations. The following examples represent the *LoadedAndOperational* situation and the policy *ReturnAndUnload*



```
CONCEPT Gripper{
  CHILDREN{}
  PARENTS{SC.Thing..Mechanics}
  ...
  FUNCS{
    FUNC lay_down_f{
      TYPE{SC.Action.Lay_Dwn}
    }
    FUNC grab_f{
      TYPE{SC.Action.Grab}
    }
    ...
  }
}
```

Figure 9: Gripper Concept and its Realization

specified to eventually handle that situation. Recall that we need also to specify a *relation* that connects these two structures if we want the policy to handle the situation (see Section 2.4.2).

```
CONCEPT_SITUATION LoadedAndOperational {
  CHILDREN {}
  PARENTS {SC.Thing..Situation}
  SPEC {
    SITUATION_STATES { SC.Thing..Robot.operational , SC.Thing..Gripper.locked_stable }
    SITUATION_ACTIONS {  SC.Action.Move ,  SC.Action.Lay_dwn }
  }
}

CONCEPT_POLICY ReturnAndUnload {
  SPEC {
    POLICY_GOAL { UnloadGripper }
    POLICY_SITUATIONS { LoadedAndOperational }
    POLICY_RELATIONS {....}
    POLICY_ACTIONS {....}
    POLICY_MAPPINGS {
      MAPPING {
        CONDITIONS {SC.Action.Get_position = B }
        DO_ACTIONS {SC.Action.Lay_dwn }
      }
      MAPPING {
        CONDITIONS {SC.Action.Get_position <> B }
        DO_ACTIONS {SC.Action.Plan_trip, SC.Action.Move  }
      }
    }
  }
}
```

4 The Pyramid of Awareness

The ultimate goal of our KR approach is to allow for awareness and self-awareness capabilities of ASCENS-like systems. In this second year of the project, in addition to the work done on KnowLang, we also developed a comprehensive conceptual model for awareness in computerized systems, which we called ”The Pyramid of Awareness” [Vas12a, VH12a]. The awareness model is presented in this section.

4.1 Awareness

In general, any autonomic system engages in interactions where it is not just able to interact with its operational environment, but also to perceive important structural and dynamic aspects of the same [KC03, VH10]. To become interaction-aware such a system needs to be aware of its physical environment and whereabouts and its current internal status. This ability is defined as awareness and it helps intelligent computerized systems to sense, draw inferences for their own behavior and react. The notion of awareness should be generally related to perception, recognition, thinking and eventually prediction. Closely related to artificial intelligence, awareness depends on the knowledge we must transfer to a computerized system and make it use that knowledge, so it can exhibit intelligence. However, in addition to computerized knowledge, artificial awareness also requires a means of sensing changes (e.g., event perception and data gathering), so the external and internal worlds can be perceived through their raw events and data. Thus, self-monitoring and monitoring the environment is the key to awareness, i.e., to exhibit awareness, computerized systems must sense and analyze their internal components and the environment where they operate. Such systems should be able to notice a change and understand its implications. Moreover, an aware system should be able to determine normal and abnormal states.

4.2 Classes of Awareness

Awareness can be classified into two major classes: self-awareness about the internal world and context awareness about the external world. The autonomic computing research [KC03] defines these two classes as following:

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

Another intriguing class of awareness could be the so-called *situational awareness*, which is related to situations. Situation awareness considers circumstances particularly relevant to important situations a computerized system can be involved in. Other classes might be more specific and draw our attention to specific problems, e.g., operational conditions and performance (operational awareness), control processes (control awareness), interaction processes (interaction awareness), navigation processes (navigation awareness), etc. Note that although classes of awareness may differ by their subject, basically they all require perception of events and data from the subjective context ”within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future” [End95].

To better understand the idea of awareness in computerized systems, we may think of an example with *exploration robots* where we may consider navigation awareness, which requires context-relative plots of position so that the system can infer robot speed and direction. Landmarks should be represented as part of the environment knowledge. Moreover, at the beginning of the navigation process a special ”navigation map” can be built on the fly by the navigation awareness mechanism. Then, basically navigation awareness is reading the sensor data from cameras and plotting the position of the robot at the time of observation. Via repeated position plots, the course and land-reference speed of the robot is established.

4.3 Structuring Awareness

Lately, there have been significant research efforts in the implementation of awareness for computerized systems. For example, commercially-available server monitoring platforms, such as NimSoft’s NimBUS and JJ Labs’ Watch Tower, offer robust, lightweight sensing and reporting capabilities across large server farms. Note that these solutions are oriented towards massive data collection and performance reporting, and leave much of the final analysis and decision-making to the administrator. In other approaches, awareness is achieved through a model-based detection and response based on offline training and models (e.g., Markov models) constructed to represent different scenarios that can be recognized by the system at runtime.

To function, the mechanism implementing awareness must be structured taking into consideration possible different stages of an awareness process. The mechanism of awareness might be built over a complex chain of functions pipelining the stages of the awareness process such as: 1) *raw data gathering*; 2) *data passing*; 3) *filtering*; 3) *conversion*; 4) *assessment*; 5) *projection*; and 6) *learning*. As shown in Figure 10, ideally all the *awareness functions* might be structured as a **Pyramid of Awareness** forming the mechanism that converts raw data (facts, measures, raw events, etc.) into conclusions, problem prediction and eventually may trigger learning.

As shown in Figure 10, the different pyramid levels represent awareness functions that can be grouped into four function groups determining specific *awareness tasks*. The first three pyramid levels compose the group of *monitoring tasks*. Further, the fourth level forms the group of *recognition tasks*. The fifth and the sixth levels compose the group of *assessment tasks*, and finally, the last seventh level form the group of *learning tasks*. In addition, *aggregation* can be included as a subtask at any function level. Note that aggregation is intended to improve the overall awareness performance, e.g., aggregation techniques can be applied to aggregate large amounts of sensory data during the filtering stage, or can be applied by the recognition tasks to improve classification.

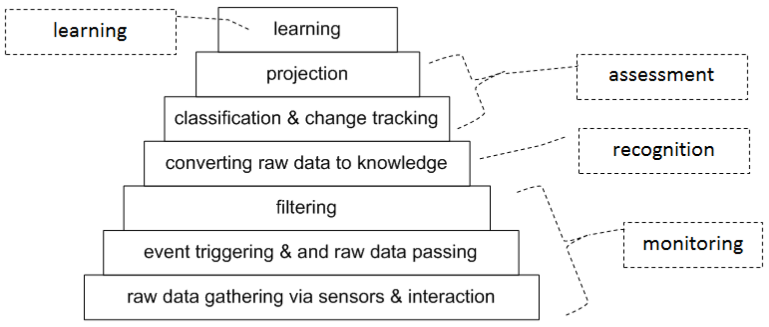

learning
learning
projection
assessment
classification & change tracking
converting raw data to knowledge
recognition
filtering
event triggering & and raw data passing
monitoring
raw data gathering via sensors & interaction

Figure 10: The Pyramid of Awareness

Ideally, the four awareness function groups require a comprehensive and well-structured KB representing knowledge in KR Symbols expressing the system itself with its proper internal structures and functionality and the environment. Moreover, the awareness process is not as straightforward as one might think. Instead, it is a cyclic with many iterations over the awareness functions. Thus, by closing the chain of awareness functions we form a special *awareness control loop* [VH10] where different classes of awareness may emerge (see Figure 11).

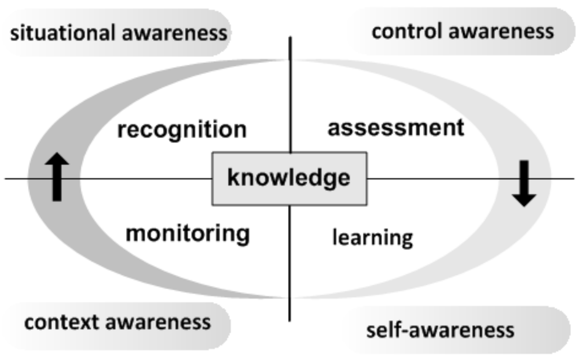


Figure 11: Awareness Control Loop

A more elaborated description of the awareness function groups is the following:

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

Its cycling nature is the main reason to regard awareness as complex product with several *levels of exhibition* and eventually *degree of awareness*. 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 and eventually interpreted with some degree of probability. Other levels of awareness exhibition might be *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. 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, an aware computerized system should rely on early awareness to react quickly to situations when fast reaction is needed and on late awareness when more precise thinking is required.

Ideally, awareness should be a part of the cognitive process where it might support learning. An efficient awareness mechanism should rely on both past experience and new knowledge introduced to the system. Moreover, awareness via learning is the basic mechanism for introducing new facts into the cognitive system - other possible ways are related to interaction with a human operator who may manually introduce new facts into the KB.

4.4 Implementing Awareness

To build an efficient awareness mechanism, we need to think how to properly integrate the Pyramid of Awareness within the KnowLang Framework. The baseline is to provide a means of *monitoring* and *knowledge representation* with proper reasoner supporting the Pyramid of Awareness. As shown in Section 2.4.1, a KR with KnowLang adds a new context to the program and the KnowLang Reasoner operates in this context taking into account the monitoring activities driven by the system's sensors and reported via TELL Operators to the reasoner (see also Figure 2). Therefore, the KnowLang Reasoner must drive the Awareness Control Loop and deliver awareness results to the system as outputs of the ASK Operators.

In addition to the awareness abilities initiated via ASK and TELL operators, we envision an additional awareness capability based on *self-initiation* where the KnowLang Reasoner may initiate actions without being asked for it. In this approach, we consider a behavior model based on the so-called Partially Observable Markov Decision Processes (POMDP) [Lit96]. Note that this model is appropriate when there is uncertainty and lack of information needed to determine the state of the entire system. For example, individuals in complex systems like swarms of robots (e.g., the ASCENS case study on marXbots robots) often might be idle, i.e., not actively participating in the swarm's activities, because they are not certain about the current swarm state. Thus, the POMDP model helps a robot reason on the current swarm state (or that of the environment) and eventually self-initiate when an action is needed to be performed. According to our POMDP-based model, a swarm robot takes as input observable situations, involving other swarm robots and the environment, and generates as output actions initiating robot activity. Note that the generated actions affect the global swarm state.

Formally, this model is a tuple $M :=< S, A, T, R, X, O >$ where:

- S is a finite set of states of the system that are not observable.
- An initial belief state $s_0 \in S$ is based on $z_0(s_0; s_0 \in S)$, which is a *discrete probability distribution* over the set of system states S, representing for each state the robot's belief that is currently occupying that state.
- A is a finite set of actions that may be undertaken by the robot. Note that the system state determines the current situation and thus, the possible set of actions is reduced to coop with that situation (or respectively state).

- $T : S \times A \longrightarrow Z(S)$ is the state transition function, giving for each system state s and robot action a, a probability distribution over states. Here, $T(s; a; s')$ computes the probability of ending in state s', given that the start state is s and the robot takes action a, $z(s'|s; a)$.
- $O : A \times S \longrightarrow Z(X)$ is the observation function giving for each system state s and robot action a, a probability distribution over observations X. For example, $O(s'; a; x)$ is the probability of observing x, in state s' after taking action a, $z(x|s'; a)$.
- $R : S \times A \longrightarrow R$ is a reward function, giving the expected immediate reward gained by the robot for taking an action in a state s, e.g., $R(s; a)$. The reward is a scalar value in the range $[0..1]$ determining, which action (among many possible) should be undertaken by the robot in compliance with the swarm goals.

Interpretation. To illustrate this model, let's assume that a marXbot swarm is currently occupying the state s = *"new object to be moved is discovered, but no moving team has been formed yet and still no other marXbot has self-initiated for team formation"*. Let's assume there is at least one idle marXbot in the swarm ready to undertake a few actions A, including the action a = *"self-initiation for team formation"*. The marXbot performs the following reasoning steps in order to self-initiate for team formation:

1. The marXbot computes its current belief state s_0 - the robot picks up the state with the highest probability z_0 and eventually $s_0 = s$.
2. The marXbot computes the probability z_1 of the swarm occupying the state s' = *"new object is discovered and a marXbot has self-initiated for team formation"* if the action a is undertaken from state s_0.
3. The marXbot computes the probability $z_2(x|s'; a)$ of observation x = *"there are sufficient numbers of idle marXbots to form a new exploration team"*.
4. The marXbot computes the reward $r(s_0; a)$ for taking the action a (self-initiation for team formation) in state s_0. If no other immediate actions should be undertaken (forced by other swarm goals), the reward r should be the highest possible, which will determine the execution of a.

Probability Computation. The POMDP model for self-initiation requires the computation of a few probability values. Our *model for assessing probability* applicable to the computation of POMDP probability values (probability of the swarm being in a state and probability of observation) is basically based on the probability distributions at the levels of *situation-policy relations* and *policy mappings* connecting conditions to actions (see Definition 19 in Section 2.1). Therefore, we need to provide the system with initial probability distributions at these levels, which practically is building Bayesian networks in our KR model. In our approach, the *probability assessment* is an indicator of the number of possible execution paths a robot may take meaning the amount of certainty (excess entropy) in the swarm's behavior. To assess that behavior prior to the KR, it is important to understand the complex interactions among the robots (SCs) in an SCE swarm. This can be achieved by modeling the behavior of individual reactive robots together with the swarm (or team) behavior as *Discrete Time Markov Chains* [EG05], and assessing the level of probability through calculating the probabilities of the state transitions in the corresponding models. We assume that the robot-swarm interaction is a stochastic process where the swarm events are not controlled by the robot and thus their probabilities are considered equal.

The theoretical foundation for our Probability Assessment Model is the property of Markov chains, which states that, given the current state of the swarm, its future evolution is independent of its history, which is also the main characteristic of a reactive autonomic robot.

Table 1: Transition Matrix Z

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

An algebraic representation of a Markov chain is a matrix (called transition matrix) (see Table 1) where the rows and columns correspond to the states, and the entry z_{ij} in the i-th row, j-th column is the transition probability of being in state s_j at the stage following state s_i. The following property holds for the calculated probabilities:

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

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

$$Z_{SCE} = \sum_{i=1,k} H_i - H$$
$$H_i = -\sum_j z_{ij} log_2(z_{ij})$$
$$H = -\sum_i v_i \sum_j z_{ij} log_2(z_{ij})$$

Here,

- H is an entropy that quantifies the level of uncertainty in the Markov chain corresponding to an SCE swarm;
- H_i is a level of uncertainty in the Markov chain corresponding to a marXbot robot;
- v is a steady state distribution vector for the corresponding Markov chain;
- z_{ij} values are the transition probabilities in the extended state machines that model the behavior of the i-th robot.

Note that for a transition matrix Z, the steady state distribution vector v satisfies the property $v{*}Z = v$, and the sum of its components v_i is equal to 1. The level of uncertainty H is exponentially related to the number of statistically typical paths in the Markov chain. Having an entropy value of 0 means that there is no level of uncertainty in a Markov system for a specific robot's behavior. Here, a higher value of a probability measure implies less uncertainty in the model, and thus, a higher level of predictability.

5 Soft Constraints for KnowLang

In general, the so-called *soft constraints* [BMR97, BM07] might be used as a KR technique that will help designers impose *constraining requirements* for special *liveness properties* of an intelligent system. In this context, the term *liveness property* must be considered as an approximation to our understanding of a good-to-have property.

In this second year of the project, we started collaborating with WP2 to integrate the theory of soft constraints in KnowLang. The goal is to enrich KnowLang with a model for KR where we can enforce desired restrictions on values held by a variety of system's variables.

With the notion of Soft Constraint for KnowLang (SCKL), we intend to associate tuples of possible values held by special KnowLang "variables" with possible preferences. Thus, to express a SCKL, we consider [MV12]:

1. a tuple of variables, representing a part of the system expressed with KnowLang;
2. a preferred combination of values held by these variables;
3. special constraint conditions defining conditions when the "preferred combinations of values" must be held.

In this approach, the notion of "variable" is closely related to the notion of KR symbol. SCKL variables (called KR Variables) represent concepts defined by the ontologies (e.g., *Policy* or *Action*), concept properties, or relations. Consecutively, the notion of "value" is associated with realization of a concept (e.g., an object), realization of a concept property, or realization of a relation. SCKL Values are also called KR Values. Note that SCKL will not be used to set what values are allowed, but rather at what specific state (component state or global system state) or situation are allowed.

5.1 Constraint Satisfaction Problem and Soft Constraints

The classical *constraint satisfaction problem* (CSP) framework [BM07] is a well-known paradigm, that is suited to specify many kinds of real-life problems and that has been broadly investigated in computer science and artificial intelligence. The key idea underlying CSP is to solve a problem by stating constraints representing requirements about the problem and, then, finding solutions satisfying all the constraints.

Some extensions of *classical* CSPs give specific interpretations of soft constraints like *weighted CSPs* for modelling cost functions, *probabilistic CSPs* or *fuzzy CSPs*. The *semi-ring-based constraints* [BMR97] are more generic extensions to soft constraints, in the sense that they can model different kinds of constraints by varying their underlying structure.

A *constraint semiring* (c-semiring) [BMR97] is an algebra $\langle A, +, \times, 0, 1\rangle$, where $\langle A, +, 0\rangle$ and $\langle A, \times, 1\rangle$ are commutative monoids, $+$ is idempotent, $\times$ distributes over $+$, 1 and 0 are absorbing elements for $+$ and $\times$ respectively (i.e., $a + 1 = 1$ and $a \times 0 = 0$ for all $a \in A$). C-semirings are also equipped with a partial ordering $\leq$ such that $a \leq b \; iff \; a + b = b$, which means that a is worse than b, or, more interestingly, that a entails b. Intuitively, the preference level associated to each variable instantiation is modelled as a value of a c-semiring; the combination of constraints is expressed by the product operation, while the sum $a+b$ chooses the worst constraint better than a and b. Moreover, 1 is the maximal and 0 the minimal element. Remarkably, several efficient algorithms defined for ordinary constraints, like constraint propagation or dynamic programming, can be generalised to c-semirings.

5.2 Elaborating on SCKL

Our notion of SCKL internalizes, in the style of [BM07], the KnowLang variables V into a c-semiring S . Namely, our SCKL constraints are functions associating to each feasible assignement of variables V a value of S. Thus to formally define a SCKL for a constraint semiring S, we consider a set of KR Variables I and a set of possible KR Values V, where for each $i \in I$, there is a set $V_i \subset V$ of possible KR Values for the variable i. A SCKL can be defined as $c := (J; P)$ with $c \in C$ (set of constraints), i.e., c is defined as a pair $(J; P)$ where $J := (j_1, ..., j_k)$ is an ordered subset of I (denoted as $J \subset I$) and P is a function mapping $V_{j1}\times, ..., \times V_{jk}$ into S.

The semiring values are ordered (usually, totally ordered) by the semiring ordering $\leq$. Thus it is possible to identify the preferred variable assignments as those with the highest semiring value associated to them.

Often, SCKLs have additional *constraint conditions* Z, expressed as Boolean operations over the KnowLang Ontologies O. Normally, a condition shall be expressed with ”states”, ”situations”, ”events”, and/or ”actions”.

Once KnowLang variables and their possible values are fixed, different SCKLs are characterized by their functions P. Thus the semiring operations of addition and multiplications can be extended to SCKLs by composing functions P pointwise.

In addition, an operation of *restriction* can be defined with eliminates a variable assigning to it the best possible value. Technically, the restriction $(x)p(x)$ is obtained by summing up $p(v)$ for all possible values of v. The *projection* operation is restriction's dual: to evidence the effect of a constraint on a set of variables is sufficient to restrict it with respect to all the other variables.

Complex *SCKL systems* can be obtained by applying the above operations to elementary (typically finite, explicitly listed) SCKLs. Often, several elementary SCKLs involving only a few variables are multiplied, obtaining a large *constraint network*. Then the resulting complex SCKL is projected into some variables, which represent the visible interface of the system.

A *Constraint Satisfaction Problem* (CSP) for a SCKL system is to find an assignment of its variables which returns the best semiring value. Of course there can be ties, i.e. several assignments can yield the same value, which makes the solution process nondeterministic. The situation is particularly critical for the classical semiring, where assignments can return only 1 or 0, i.e. possible or impossible. If the semiring ordering is partial, a solution is an assignment which returns a semiring value which is non-dominated (i.e. it is a local maximum, sometimes called *Pareto-optimal*).

5.3 Soft Constraints for marXbot Robot

With reference to the KR structures presented in Section 3, we may consider the following constraint for marXbot. A marXbot robot has six wheels, each of which can be considered as a variable that can take state values in:

```
wheel.STATES := { clockwise, counterclockwise, stop, idle }
```

The wheels operate in pairs, and on each wheel pair we apply a constraint stating that pair weels must take ”certain” values. In fact, if two wheels in a pair are turning clockwise and anticlockwise respectively the pair is moving forward. If in a pair a wheel is turning clockwise and the other wheel is stopped, then the pair turns left, etc. Here, the tree pairs of wheels can have global state values as following:

```
wheel_pair.STATES := {forward, backward, left, right, stop, idle}
```

The soft constraint approach considers a pair of wheels equipped with three variables representing left and right wheels and the global state of the pair:

```
pair(global) :- left(global,leftwheel),right(global,rightwheel)
```

where the constraints $left$ and $right$ determine the relation between the possible states of the wheels and the global state. For classical Horn clauses, the relational operator ”:-” means that the right hand side implies the left hand side, while for all the soft constraints ”:-” means that the right hand side is larger than the left hand side in the semiring partial ordering. The comma ”,” means multiplication, logical AND in the classical case. Here, for a pair, $left$ and $right$ are defined as:

```
left(forward,counterclockwise) :-
right(forward,clockwise) :-
left(backward,clockwise) :-
right(backward,counterclockwise) :-
left(left,stop) :-
right(left,clockwise) :-
left(right,counterclockwise) :-
right(right,stop) :-
left(idle,idle) :-
right(idle,idle) :-
left(stop, stop) :-
right(stop,stop) :-
```

Here the empty right hand side obviously means 1. For instance, the assignment $global := forward$, $leftwheel := counterclockwise$ and $rightwheel := clockwise$ is allowed, namely it is mapped to 1. Conversely, no assignment is possible with $leftwheel := clockwise$ and $rightwheel := clockwise$.

The robot has three pairs of wheels and a GLOBAL state variable with possible values:

```
robot[1].STATES := \{forward, left, right, stop\}
```

Thus:

```
robot[1](GLOBAL) :- OK(GLOBAL,global1,global2,global3),
                    pair(global1),pair(global2),pair(global3)
```

where OK is defined as:

```
OK(forward,forward,forward,forward) :- very fast
...
OK(forward,forward,idle,idle) :- slow
OK(left,left,idle,idle) :- slow
...
```

Notice that here OK has been defined in a soft way: the option where the forward action of the robot is obtained with all three pairs contributing to it has been evaluated as ”very fast”, assuming that the semiring has such a value, while if only one pair has forward and the other two have idle, the value is ”slow”. Here the idea is that ”very fast” is better in the semiring ordering than ”slow”.

6 KnowLang vs SCEL and SOTA

Ideally, KnowLang is going to be used to build KR models that must be integrated in the SCEs implemented with SCEL, the Service Component Ensembles Language tackled by WP1. KnowLang should provide a KR model of the SCEL knowledge base. Therefore, the KR, built with KnowLang, must cope with the system implementation specified with SCEL. The KR models are going to be used by the system (programmed with SCEL) to find missing answers, e.g., find behavior that will help solve a problem that is not initially programmed. Recall that both the KnowLang concepts and objects might be specified to represent entities implemented in the system's program (see Section 2.2.1). Moreover, these KR models must be kept relevant all the time during the system's execution. This can be achieved by updating the KB with the most recent errors, events and actions raised/executed in the system or in the environment (see Section 2.4.1). The points where KnowLang and SCEL must work together are:

1. The ASK and TELL KB Operators (see Section 2.4.1) shall be implemented by the KnowLang Reasoner, but also should be interfaced in the system's implementation as well. This includes work on parameters passing and work on parameters and results mapping. Recall that both ASK and TELL Operators may pass parameters to the KB, which must be mapped to KnowLang's symbols before processing the call within the KR Context (see Section 2.4). Moreover, the ASK

Operators return a result produced by the KnowLang Reasoner and this result must be mapped to implemented variables and structures.

2. The KB Operators should be compatible with the SCEL operators for reading from and writing to the SCEL tuple space. Eventually, the KB will be stored in a SCEL tuple space where the physical location of the possibly distributed data will be transparent.

3. The ontologies specified with KnowLang must incorporate some parts of the SCEL description of the ASCENS system under development. This will help the developers map concepts and objects specified with KnowLang to program structures specified with SCEL. For example, KnowLang policies may address a particular instance of SCEL policies dedicated to adaptation.

To provide for a basic knowledge representation mechanism and knowledge handling, SCEL eventually relies on DEECo presented in D1.5 [BGH$^+$12]. However, to increase its expressiveness and extend its knowledge handling capabilities, DEECo shall incorporate both the KnowLang specification model and the KnowLang Reasoner where the TELL and ASK operators should also be integrated.

KnowLang will be used to model situations and self-adaptation policies determined with SOTA, the State Of The Affairs framework tackled by WP4. SOTA can be used to produce a set of requirements and guidelines for knowledge modeling and representation, which will be then specified with KnowLang. Moreover, WP4 has compiled an extensive catalogue of *adaptation patterns* [ZAC$^+$12] focused on the relations among different SCs within a shared ensemble. Considering that KnowLang has been designed to keep track of the system and the environment (including the topology of the network), these patterns could be expressed in KnowLang to provide for predefined self-adaptation scenarios.

A part of the KnowLang states can be derived from the General Ensemble Model(GEM)'s structures defining the state space as the result of an interaction between the ensemble and its environment. Also, KnowLang policies can be built to impose behavior based on POEMs strategies. Note that GEM is a mathematical model for the behavior of ensembles in the *state-of-the-affairs* space, and the Pseudo-Operational Ensemble Modeling Language (POEM) is a modeling language based on GEM (and thus SOTA) [WHM$^+$12].

7 Summary and Future Goals

In the course of the second year of WP3, we shaped our research activities towards focusing on the KnowLang Framework where our ultimate goal is to structure computerized knowledge so that a computerized system can effectively process it and gain awareness capabilities and eventually derive its own behavior. To provide comprehensive and powerful specification formalism, we developed a powerful multi-tier specification model where ontologies are integrated with rules and Bayesian networks. The approach allows for efficient and comprehensive knowledge structuring and awareness based on logical and statistical reasoning. We used the KnowLang notation to specify some initial knowledge models for all three ASCENS case studies. Although, those models need to be shaped and developed further, this exercise demonstrated the ability of KnowLang to handle KR for systems from different application domains. In addition, along with further development of the language theory and knowledge-specification structures, we started working on the KnowLang Reasoner and started implementing the KnowLang Toolset. A very important milestone we overcame is the KnowLang mechanism for self-adaptive behavior where knowledge representation and reasoning help to establish the vital connection between knowledge, perception, and actions realizing self-adaptive behavior. The knowledge is used against the perception of the world to generate appropriate actions in compliance to some goals and beliefs. To support this approach, we developed a KR mechanism for self-adaptive behavior and started developing special ASK and TELL operators used by the system to talk to the KnowLang Reasoner. We developed an initial operational semantics for these operators. Moreover, we developed a conceptual reference model for awareness called ”Pyramid of Awareness” and outlined how this model can be realized with the KnowLang Framework. Finally, a joint work with WP2 derived a theoretical model for Soft Constraints for KnowLang where the goal is to help designers impose constraining requirements for special liveness properties.

Our plans for the third year of WP3 are mainly concerned with further development of KnowLang. As part of Task T3.1, we shall complete the formal notation and implement appropriate tools for KnowLang, such as Grammar Compiler (partially completed), Visual and Textual Editor (partially completed), Syntax and Consistency Checker and KnowLang Specification Compiler.

As for Task 3.2, we shall continue with further development of the KR models for the three ASCENS Case Studies. This eventually will help us derive generic KR models for ASCENS-like systems.

Task 3.3, will continue with further development of the KnowLang Reasoner where we need to complete the KB Operators and start implementing the Inference Primitives together with the KnowLang Awareness Mechanism.

Finally, in Year 3 of the project, we shall start Task 3.4 with work on awareness prototyping.

References

[BGH$^+$12] T. Bures, I. Gerostathopoulos, V. Horky, J. Keznikl, J. Kofron, M. Loreti, and F. Plasil. D1.5: Language Extensions for Implementation–Level Conformance Checking, 2012. ASCENS Deliverable.

[BL04] R.J. Brachman and H.J. Levesque. *Knowledge Representation and Reasoning*. Elsevier, New York, 2004.

[BM07] M. G. Buscemi and U. Montanari. CC-Pi: a constraint-based language for specifying service level agreements. In R. De Nicola, editor, *ESOP 2007, LNCS 4421*, pages 18–32. Springer, 2007.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[BN03] F. Baader and W. Nutt. Basic description logics. In *The Description Logic Handbook (ed. F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider*, pages 43–95. Cambridge University Press, Cambridge, UK, 2003.

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

[End95] M. R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors*, 37(1):32–64, 1995.

[GFMGS08] C. Galindo, J. Fernandez-Madrigal, J. Gonzalez, and A. Saffiotti. Robot task planning using semantic maps. *Robotics and Autonomous Systems*, 56(11):955–966, 2008.

[Hal90] J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1990.

[HBGK12] M. Holzl, L. Belzner, T. Gabor, and A. Klarl. D8.2: Second Report on WP8. The ASCENS Service Component Repository (first version), 2012. ASCENS Deliverable.

[HNW08] H. Holzapfel, D. Neubig, and A. Waibel. A dialogue approach to learning object descriptions and semantic categories. *Robotics and Autonomous Systems*, 56(11):1004–1013, 2008.

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KLB^{+}07] G.-J. M. Kruijff, P. Lison, T. Benjamin, H. Jacobsson, and N. Hawes. Incremental, multi-level processing for comprehending situated dialogue in human-robot interaction. In *Proceedings of the Symposium on Language and Robots*, 2007.

[Knu64] D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736, 1964.

[Lit96] M. L. Littman. Algorithms for sequential decision making, 1996. PhD Thesis, Department of Computer Science, Brown University.

[MJZ^{+}07] O. Mozos, P. Jensfelt, H. Zender, G.-J. M. Kruijff, and W. Burgard. An integrated system for conceptual spatial representations of indoor environments for mobile robots. In *Proceedings of the IROS 2007 Workshop: From Sensors to Human Spatial Concepts (FS2HSC)*, pages 25–32, 2007.

[MV12] U. Montanari and E. Vassev. Soft constraints for KnowLang. In *Proceedings of C* Conference on Computer Science & Software Engineering (C3S2E '12)*, pages 99–103. ACM, 2012.

[Nea03] R. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, 2003.

[Vas09] E. Vassev. *ASSL: Autonomic System Specification Language - A Framework for Specification and Code Generation of Autonomic Systems*. LAP Lambert Academic Publishing, November 2009.

[Vas12a] E. Vassev. Building the pyramid of awareness. *Awareness Magazine - Self-awareness in Autonomic Systems*, July 2012.

[Vas12b] E. Vassev. Initial knowledge representation models for the ASCENS case studies. Technical Report Lero-TR-2012-06, Lero, University of Limerick, Ireland, 2012.

[Vas12c] E. Vassev. KnowLang grammar in BNF. Technical Report Lero-TR-2012-04, Lero, University of Limerick, Ireland, 2012.

[Vas12d] E. Vassev. Operational semantics for KnowLang ASK and TELL operators. Technical Report Lero-TR-2012-05, Lero, University of Limerick, Ireland, 2012.

[VH10] E. Vassev and M. Hinchey. The challenge of developing autonomic systems. *IEEE Computer*, 43(12):93–96, 2010.

[VH11] E. Vassev and M. Hinchey. Towards a formal language for knowledge representation in autonomic service-component ensembles. In *Proceedings of the 3rd International Conference on Data Mining and Intelligent Information Technology Applications (ICMIA2011)*, pages 228–235. AICIT, IEEE Xplore, 2011.

[VH12a] E. Vassev and M. Hinchey. Awareness in software-intensive systems. *IEEE Computer*, 45(12), 2012.

[VH12b] E. Vassev and M. Hinchey. Efficient reasoning with ambient trees for space exploration. In *Proceedings of the International Conference on Context-Aware Systems and Applications (ICCASA 2012)*. Lecture Notes of ICST (LNICST), 2012.

[VH12c] E. Vassev and M. Hinchey. Knowledge representation for cognitive robotic systems. In *Proceedings of the 15th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing Workshops (ISCORCW 2012)*, pages 156–163. IEEE Computer Society, 2012.

[VH12d] E. Vassev and M. Hinchey. Knowledge representation with KnowLang - the marXbot case study. In *Proceedings of the 11th IEEE International Conference on Cybernetic Intelligent Systems (CIS 2012)*. IEEE Computer Society, 2012.

[VHng] E. Vassev and M. Hinchey. Knowledge representation and reasoning for self-adaptive behavior and awareness. *TCCI - Special Issue on ICECCS 2012*, 2013 (pending).

[VHG$^+$11] E. Vassev, M. Hinchey, B. Gaudin, P. Nixon, N. Bicocchi, and F. Zambonelli. D3.1: First Report on WP3. Software requirements, knowledge modeling and knowledge representation for self-awareness - report and survey with experimental results for intelligent multi-agent systems, 2011. ASCENS Deliverable.

[VHG12] E. Vassev, M. Hinchey, and B. Gaudin. Knowledge representation for self-adaptive behavior. In *Proceedings of C* Conference on Computer Science & Software Engineering (C3S2E '12)*, pages 113–117. ACM, 2012.

[WHM$^+$12] M. Wirsing, M. Hoelzl, U. Montanari, E. Vassev, F. Zambonelli, M. Loreti, F. Tiezzi, S. Bensalem, R. Bruni, A. Lluch Lafuente, R. Pugliese, and R. De Nicola. JD2.1: Languages and knowledge models for self-awareness and self-expression, 2012. ASCENS Deliverable.

[ZAC$^+$12] F. Zambonelli, D.B. Abeywickrama, G. Cabri, M. Puviani, M. Hoelzl, A. Corradini, A.L. Lafuente, and R. De Nicola. D4.2: Second Report on WP4. Component- and ensemble-level self-expression patterns: Report on experimental and simulation activities, and requirements for tools implementation, 2012. ASCENS Deliverable.