www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D3.4: Fourth Report on WP3

Knowledge Representation and Awareness with KnowLang

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 3.0 (29.4.2014)**

Lead contractor for deliverable: **UL**
Author(s): **Emil Vassev (UL), Mike Hinchey (UL), Nicklas Hoch (VW), Henry P. Bensler (VW)**

Reporting Period: **4**
Period covered: **October 1, 2013 to March 31, 2015**
Submission date: **March 12, 2015**
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

This report outlines the last year of Research & Development within WP3 and concludes the overall results achieved within this WP's mandate. WP3 was initially intended to handle the problem of knowledge representation (KR) and awareness of self-adaptive systems. In the first three years of the project, we developed the KnowLang Framework to provide both notation and tools for knowledge representation. In the last 4th year of the project, we developed the KnowLang Reasoner, which complements our work on the KnowLang Framework by providing a reasoner operating over the KnowLang-specified and KnowLang-compiled knowledge bases (KBs). The KnowLang Reasoner is an efficient and powerful reasoning mechanism that is meant to support reasoning about self-adaptive behavior. It runs as a component in host, self-adaptive systems such as the ASCENS ensembles. The reasoner communicates with a host system via special ASK and TELL operators and operates in the KR context outlined by the KnowLang-compiled KB, a context formed by the represented knowledge. The TELL operators feed the KR context with important information driven by errors, executed actions, new sensory data, etc., thus helping the KnowLang Reasoner update the KB 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 a current situation involving the environment, and ideally with behavior models for self-adaptation.

In this deliverable, we report on our continuous work on specifying KR models for the ASCENS Case Studies. In particular, we report our results on building the KR models for eMobility and Swarm Robotics. The second part of this deliverable outlines our work on the KnowLang Reasoner. In particular, we present the design and implementation of the KnowLang Reasoner, along with test results of using the reasoner to simulate awareness for the eMobility ASCENS Case Study. Similar to the previous years, in this year, we continued collaborating with the other WPs taking part in the ASCENS Project, mainly with WP7, WP4, and WP8.

Contents

1 Introduction

One of the major breakthroughs of the ASCENS Project, is KnowLang, which includes both the KnowLang Framework and KnowLang Reasoner. During the project's four years of research & development, WP3 has accomplished results allowing engineers of self-adaptive systems to build knowledge models with built-in adaptive mechanisms, and use these models at runtime to control the behavior of self-adaptive systems in critical situations requiring self-adaptation. In this deliverable, we outline the results achieved by WP3 during the last, forth year of the ASCENS Project.

1.1 Research & Development in Year 4

In the last, fourth year of the project, after the successful completion of the KnowLang Framework (WP3.T1), we continued working on the knowledge models for the ASCENS Case Studies (WP3.T2), but the focus of our major efforts was put on the implementation of the KnowLang Reasoner (WP3.T3) and using the latter to perform simulation of awareness for ASCENS ensembles (WP3.T4). At the time of this document's writing, we have implemented in KnowLang the knowledge bases (KBs) for the eMobility and Swarm Robotics case studies, and with the completion of the KB for the Science Clouds case study in Year 3 of the project [VHBM13], we concluded the task WP3.T2. Moreover, we have implemented the first version of the KnowLang Reasoner. Although still limited in terms of its learning abilities, currently, the reasoner can efficiently operate over all the knowledge models we have specified with KnowLang. With this, we completed the task WP3.T3. Finally, we have used the implemented KnowLang Reasoner, along with a test application, hosting that reasoner, to perform awareness tests where we tested against the eMobility knowledge model, which concludes the task WP3.T4.

1.2 Relations with Other WPs

In the fourth year of this project, we continued collaborating intensively with WP1, WP2, WP4, WP7, and WP8. The collaboration with WP1 was at the level of interoperability between SCEL and the KnowLang Reasoner. KnowLang provides a KR model of the SCEL knowledge base and the Knowlang Reasoner should be properly integrated with SCEL. We also worked on the integration of POEM in the reasoning process driven by the KnowLang Reasoner. Although not fully accomplished, this work paved the way of using POEM as a FOL reasoner in the reasoning process driven by the KnowLang Reasoner.

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

Concerning WP4, in this third year of the project, we worked together on the problem of *situational awareness and reasoning*. For data collection for awareness purposes, we used of a recently-developed framework [BFZ14], centered around the concept of dynamic service reconfiguration, which is able to gather data from a number of different sources, and classify them using general-purpose algorithms. For reasoning, we used the KnowLang Framework and the Knowlang Reasoner. This joint work resulted in the joint publication [BVZH14].

WP7 [SMP+12, SHP+13] provides vital experimental platforms for both the notation and toolset of KnowLang. In collaboration with WP7, this year, we used again the Autonomy Requirements Developed (ARE) approach [VH13a] to capture relevant knowledge data and then we used KnowLang to specify a complete, yet relevant knowledge models for both eMobility and Science Clouds case studies. In these endeavor, WP7 provided us with important information related to the possible state expressions and scenarios. This joint work also resulted in two publications [VHBH14, VH14].

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

1.3 Structure of the Document

The rest of this document is organized as follows. Section 2 provides a brief overview of the ARE approach, along with an overview of the KnowLang mechanism for specifying self-adaptive behavior. This section provides a required background for the rest of the document. Sections 3 and 4 present the KnowLang knowledge specification models for the eMobility and Swarm Robotics case studies respectively. In Section 5 we outline the implementation of the KnowLang Reasoner, along with the test results of simulating awareness for ASCENS ensembles. The presented results outline our experiments with the KnowLang Reasoner operating over the eMobility case study. Finally, to conclude the topic, in Section 6, we present a brief summary and future goals.

2 Capturing Requirements for Self-adaptive Behavior

The self-adaptive behavior is what makes the difference in self-adaptive systems. In this endeavor, we have been striving to capture this very behavior, so it can be properly designed and consecutively, implemented. To do so, we consider that self-adaptive behavior extends upstream the regular objectives of a system with special self-managing objectives, also called self-* objectives [VH13a]. Basically, the self-* objectives provide autonomy features in the form of system's ability to automatically discover, diagnose, and cope with various problems. This ability depends on the system's degree of autonomicity, quality and quantity of knowledge, awareness and monitoring capabilities, and quality characteristics such as adaptability, dynamicity, robustness, resilience, and mobility. The approach for capturing all these requirements is called Autonomy Requirements Engineering (ARE) [VH13a, VH13d, VH13c, VH13b]. The ARE approach was presented in our 3rd deliverable [VHBM13] where we demonstrated in detail the process of capturing autonomy requirements for the ASCENS Science Clouds Case Study. In this section we provide a brief overview of ARE as a needed background for the following sections explaining the formalization of the other two ASCENS Case Studies: eMobility and Swarm Robotics.

2.1 The Autonomy Requirements Engineering Approach

ARE strives to provide a complete and comprehensive solution to the problem of autonomy requirements elicitation and specification. Note that the approach targets exclusively the self-* objectives as the only means to explicitly determine and define autonomy requirements. Thus, it is not meant to handle the regular functional and non-functional requirements of the systems, presuming that those might by tackled by the traditional requirements engineering approaches, e.g., use case modeling, domain modeling, constraints modeling, etc. Hence, functional and nonfunctional requirements might be captured by the ARE approach only as part of the self-* objectives elicitation.

The ARE approach starts with the creation of a goals model that represents system objectives and their interrelationships for the system in question. For this, we use GORE (Goal-Oriented Requirements Engineering) where ARE goals are generally modeled with intrinsic features such as type, actor, and target, with links to other goals and constraints in the requirements model. Goals models might be organized in different ways copying with the system's specifics and engineers' understanding about the system's goals. Thus we may have hierarchical structures where goals reside different level of granularity and concurrent structures where goals are considered as being concurrent to each other.

The next step in the ARE approach is to work on each one of the system goals along with the elicited environmental constraints to come up with the self-* objectives providing the autonomy requirements for this particular system's behavior. In this phase, we apply a special Generic Autonomy Requirements model to a system goal to derive autonomy requirements in the form of goal's supportive and alternative self-* objectives along with the necessary capabilities and quality characteristics.

Finally, the last step after defining the autonomy requirements per system's objectives is the formalization of these requirements, which can be considered as a form of formal specification or requirements recording. The formal notation used to specify the autonomy requirements is KnowLang [VHM+12, VHBM13, VH15]. Recall that the process of requirements specification with KnowLang goes over a few phases:

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

2. Behavior definition - identifies situations and behavior policies as "control data" helping to identify important self-adaptive scenarios.

3. Knowledge structuring - encapsulates domain entities, situations and behavior policies into KnowLang structures like concepts, properties, functionalities, objects, relations, facts and rules.

2.2 Formalizing Self-adaptive Behavior with KnowLang

To specify self-* objectives with KnowLang, we use special policies associated with goals, special situations, actions (eventually identified as system capabilities), metrics, etc.[VHM+12, VHBM13, VH15]. Hence, self-* objectives are represented as policies describing at an abstract level what the system will do when particular situations arise. The situations are meant to represent the conditions needed to be met in order for the system to switch to a self-* objective while pursuing a system goal. Note that the policies rely on actions that are a-priori-defined as functions of the system. In case, such functions have not been defined yet, the needed functions should be considered as autonomous functions and their implementation will be justified by the ARE's selected self-* objectives.

According to the KnowLang semantics, in order to achieve specified goals (objectives), we need to specify policy-triggering *actions* that will eventually change the system states, so the desired ones, required by the goals, will become effective [VHM+12, VH15]. Note that KnowLang policies allow the specification of autonomic behavior (autonomic behavior can be associated with self-* objectives), and therefore, we need to specify at least one policy per single goal, i.e., a policy that will provide the necessary behavior to achieve that goal. Of course, we may specify multiple policies handling same goal (objective), which is often the case with the self-* objectives and let the system decides which policy to apply taking into consideration the current situation and conditions. The following is a formal presentation of a KnowLang policy specification [VHM+12].

Policies (Π) are at the core of autonomic behavior (autonomic behavior can be associated with autonomy requirements). A policy π has a goal (g), policy situations (Si_π), policy-situation relations

(R_π), and policy conditions (N_π) mapped to policy actions (A_π) where the evaluation of N_π may eventually (with some degree of probability) imply the evaluation of actions (denoted with $N_\pi \overset{[Z]}{\rightarrow} A_\pi$ (see Definition 2). A condition is a Boolean function over ontology (see Definition 4), e.g., the occurrence of a certain event.

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

Def. 2 $\pi :=< g, Si_\pi, [R_\pi], N_\pi, A_\pi, map(N_\pi, A_\pi, [Z]) >$
$A_\pi \subset A, N_\pi \overset{[Z]}{\rightarrow} A_\pi$ *(A_π - Policy Actions)*
$Si_\pi \subset Si, Si_\pi := \{si_{\pi_1}, si_{\pi_2}, ..., si_{\pi_n}\}, n \geq 0$
$R_\pi \subset R, R_\pi := \{r_{\pi_1}, r_{\pi_2}, ..., r_{\pi_n}\}, n \geq 0$
$\forall r_\pi \in R_\pi \bullet (r_\pi :=< si_\pi, [rn], [Z], \pi >), si_\pi \in Si_\pi$
$Si_\pi \overset{[R_\pi]}{\rightarrow} \pi \rightarrow N_\pi$

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

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

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

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

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

Def. 8 $si :=< s, A \overset{\leftarrow}{_{si}}, [E \overset{\leftarrow}{_{si}}], A_{si} >$ *(Situation)*
$A \overset{\leftarrow}{_{si}} \subset A$ *($A \overset{\leftarrow}{_{si}}$ - Executed Actions)*
$A_{si} \subset A$ *(A_{si} - Possible Actions)*
$E \overset{\leftarrow}{_{si}} \subset E$ *($E \overset{\leftarrow}{_{si}}$ - Situation Events)*

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

A goal g is a desirable transition to a state or from a specific state to another state (denoted with $s \Rightarrow s'$) (see Definition 5). A state s is a Boolean expression over ontology ($be(O)$)(see Definition 6), e.g., "a specific property of an object must hold a specific value". A situation is 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 8).

Ideally, policies are specified to handle specific situations, which may trigger the application of policies. A policy exhibits a behavior via actions generated in the environment or in the system itself. Specific conditions determine, which specific actions (among the actions associated with that policy - see Definition 2) shall be executed. These conditions are often generic and may differ from the situations triggering the policy. Thus, the behavior not only depends on the specific situations a policy is specified to handle, but also depends on additional conditions. Such conditions might be organized in a way allowing for synchronization of different situations on the same policy. When a policy is applied, it checks what particular conditions are met and performs the mapped actions (see $map(N_\pi, A_\pi, [Z])$) - see Definition 2). An optional probability distribution can additionally restrict the action execution. Although initially specified, the probability distribution at both mapping and relation levels is recomputed after the execution of any involved action. The re-computation is based on the consequences of the action execution, which allows for reinforcement learning.

3 Formalizing eMobility with KnowLang

An eMobility system [SRA+11, SMP+12] needs to thematically, temporally and spatially coordinate mobility entities where the system must be modeled as a heterogeneous system composed of intelligent and self-aware nodes, which are cross-connected by information and communication technology. In such a system, the e-vehicles are competing for infrastructure resources of the traffic environment where the infrastructure resources are constrained. For example, roads, parking lots and charging stations have a limited capacity. The cost for a e-vehicle to use the infrastructure capacity is variable and changes with time and location. Situations occur, in which the availability of infrastructure resources does not match the demand.

eMobility brings most of the challenges that the theories and methodologies developed for self-adaptive systems are striving to solve. Hence, self-adaptation emerged as an important paradigm making eMobility capable of modifying the system behavior and/or structure in response to increasing workload demands and service failures. A common characteristic of self-adaptive eMobility is to emphasize self-adaptations required to ensure that services will be provided in a fail-safe manner and under consideration of system goals.

In eMobility vehicles move according to a schedule defined by a driver [SRA+11, SMP+12]. Every e-vehicle component is responsible for driving along the optimal route, meeting time constraints imposed by the driver's schedule and reserving spaces at a particular Point of Interest (POI). Vehicles are competing for infrastructure resources of the traffic environment and a set of locally optimal solutions should be computed for each individual driver. Each e-vehicle is equipped with a Vehicle Planning Utility (Route Planner) that plans travels including a set of alternative routes. Traffic routes are composed of multiple driving locations, e. g., POIs. A set of locally optimal solutions is computed for each individual user. This set is negotiated on a global level in order to satisfy the global perspective. The set of locally optimal solutions guarantees a minimum quality for each individual driver. The global optimization scheme guarantees optimal resource distribution within the local constraints. The size of the set of locally optimal solutions determines the cooperative nature of the individual driver. The smaller the set, the more competitive the driver is. The larger the set the more cooperative the driver is. The process of Route Selection (RouteSAM) advises on a route choice, which is made from a set of alternative routes generated by the route planner. The RouteSAM considers road capacity and traffic levels. It optimizes overall throughput of the roads by balancing the route assignments of the vehicles. From a local vehicle perspective the journey time is minimized, from a global perspective, the congestion levels are minimized. The route selection process strives to satisfy global optimality criteria of road capacity. Once a vehicle is in the close vicinity of a destination, it computes a set of locally optimal parking lots. Again, the selection process of parking lots satisfies global optimality criteria of parking capacity.

Figure 1 shows a formal petri net representation of a real example scenario that considers four

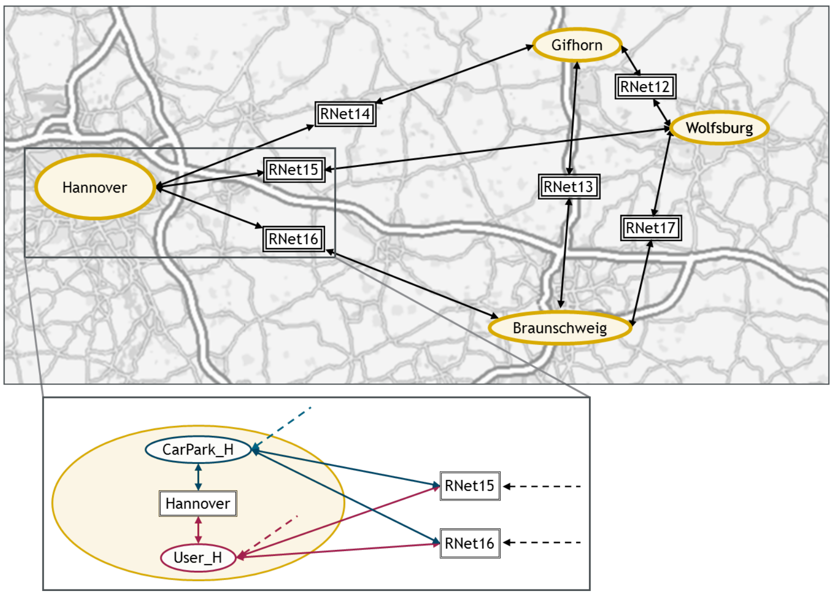


Figure 1: eMobility Example [SRA+11]

destinations (Wolfsburg, Gifhorn, Braunschweig, and Hannover), the road network between the destinations and the processes which are taking place at the destination locations [SRA+11]. The road network is described by several transition framed sub nets (e.g. RNet15). It is assumed that the journeys between destinations contain a limited set of variants. Typically three alternative routes and three alternative driving styles are considered, generating a set of maximally 9 variants. Each destination is represented by a transition framed subnet (e.g. Hannover), which models both the vehicle charging process (e.g. CarPark H) and user specific processes (e.g. User H) such as appointments. The charging stations that are connected to the car parks support three different charging modes (normal, fast and ultra-fast charging).

3.1 ARE for eMobility

In the eMobility operational environment, self-adaption is required by situations that occur when the availability of infrastructure resources does not match the demand - not enough capacity, or environment constraints (e.g., speed limit, or delay due to high traffic) hinder the e-vehicle goals. eMobility considers five different levels of self-adaptation [SHP+13]:

- *Level-1*: A vehicle computes a set of alternative routes for its current destination. This operation is performed locally by the use of the vehicle's planning utility.
- *Level-2*: A vehicle chooses the best option from those alternatives that are computed in the previous level. The vehicle observes the situation and adapts by triggering a new adaptation cycle, starting at Level-1 to the changes in the environment. This operation may require central planning and reasoning at group (ensemble) level.
- *Level-3*: A vehicle computes a set of parking lots nearby the current destination. This operation is local and is performed by the vehicle's planning utility.
- *Level-4*: A central parking lot planner (PLCSSAM) chooses the best option from those alternatives that are provided by the vehicle in the previous level. As a result vehicles are assigned an

optimal or near-optimal parking lot reservation. At the same time, a ”near-optimal parking lot” load balancing is established.

- *Level-5*: A vehicle issues a reservation request to the selected parking lot. As a result the parking space at that parking lot is booked. Both the vehicle and the parking lot monitor the situation. If required, a new adaptation cycle is triggered.

Based on the rationale above, we applied the ARE approach (see Section 2.1) and derived the eMobility goals along with the self-* objectives assisting these goals when self-adaptation is required. Figure

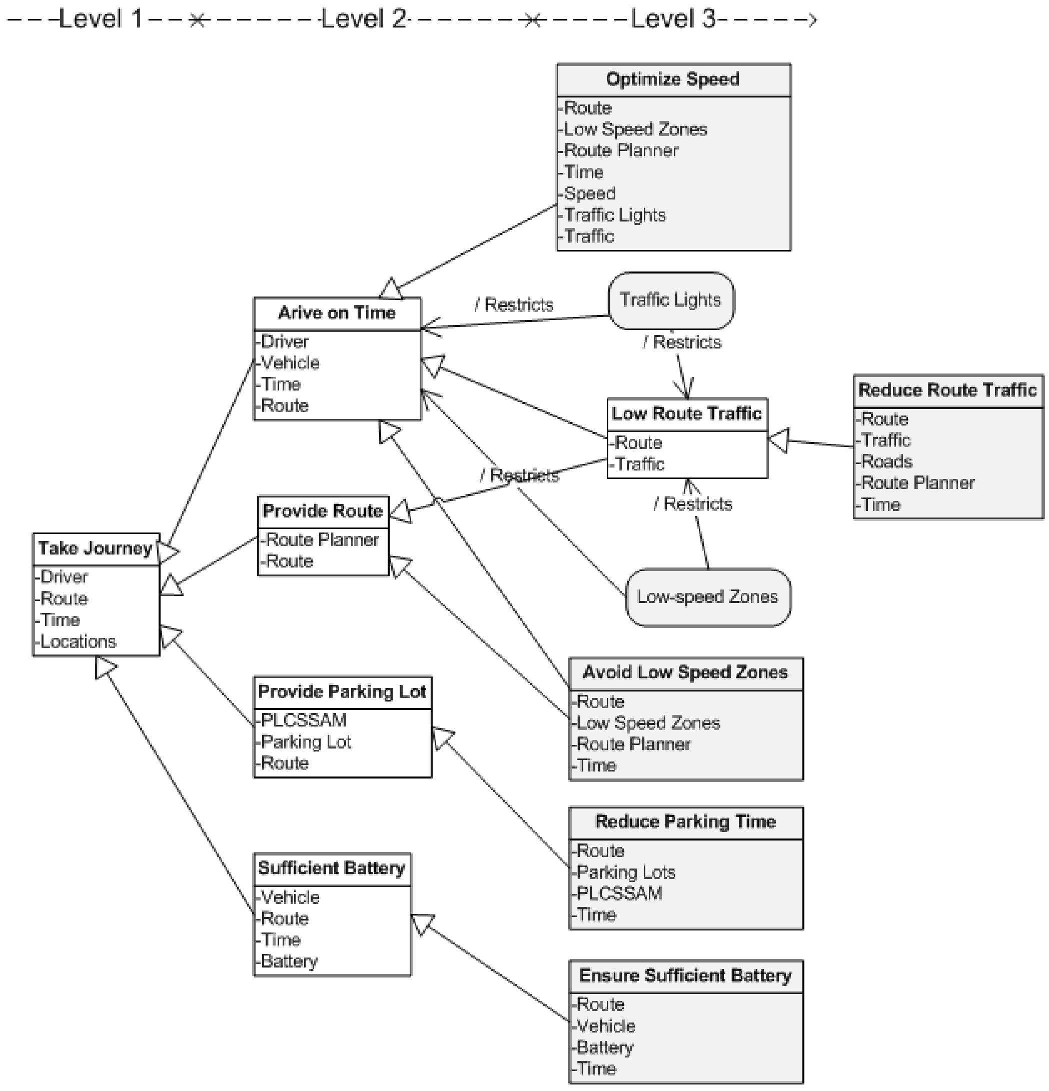


Figure 2: eMobility Goals Model with Self-* Objectives for System Goals from Level 3

2 depicts the ARE goals model for eMobility where goals are organized hierarchically at four different levels. As shown, the goals from the first two levels (e.g., ”Take Journey”, ”Arive on Time”, ”Provide Route”, ”Provide Parking Lot”, and ”Sufficient Battery”) are *main system goals* captured at different levels of abstraction. The 3rd level is resided by *self-* objectives* (e.g., ”Optimize Speed”, ”Avoid Low Speed Zones”, ”Reduce Parking Time”, and ”Ensure Sufficient Battery”) and *supportive goals* (e.g., ”Low Route Traffic”) associated with and assisting the 2nd-level goals. Finally, the goals from the 4th level are self-* objectives (e.g., ”Reduce Route Traffic”) assisting the supportive goals from the

3rd level. Basically, all the self-* objectives inherit the system goals they assist by providing behavior alternatives with respect to these system goals. The eMobility system switches to one of the assisting self-* objectives when alternative autonomous behavior is required (e.g., a vehicle needs to avoid low-speed zones). In addtion, Figure 2 depicts some of the environment constraints (e.g., ”Traffic Lights” and ”Low-speed Zones”), which may cause self-adaptation.

3.2 Specifying eMobility Ontology

In order to specify the autonomy requirements for eMobility, the first step is to specify a knowledge base (KB) representing the eMobility system in question, i.e., e-vehicles, parking lots, routes, traffic lights, etc. To do so, we need to specify ontology structuring the knowledge domains of eMobility. Note that these domains are described via domain-relevant concepts and objects (concept instances) related through relations. Recall that in order to handle explicit concepts like situations, goals, and policies, we grant some of the domain concepts with explicit state expressions where a state expression is a Boolean expression over the ontology (see Definition 6 in Section 2.2).

Figure 3, depicts a graphical representation of the eMobility ontology relating most of the domain concepts within an eMobility system. Note that the relationships within a concept tree are ”is-a” (inheritance), e.g., the RoadElement concept is a TraficEntity and the Action concept is a Knowledge and consecutively Phenomenon, etc. Most of the concepts presented in Figure 3 were derived from the eMobility Goals Model (see Figure 2). Other concepts are considered as explicit and derived from the KnowLang's specification model [VHM+12].

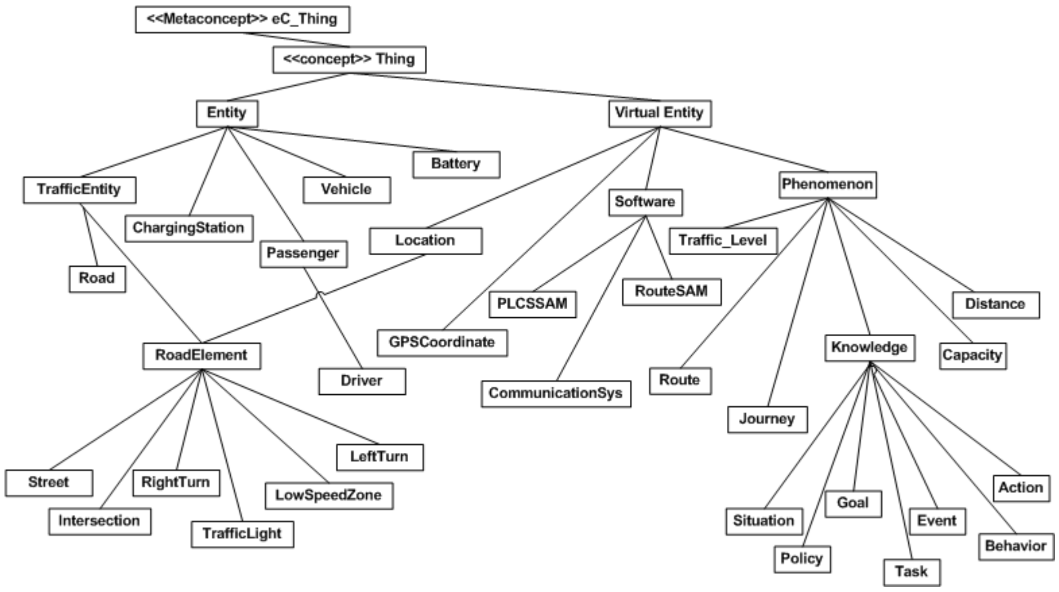


Figure 3: eMobility Ontology Specified with KnowLang

The following is a sample of the KnowLang specification representing three important concepts: $Vehicle$, $Journey$, and $Route$. As specified, the concepts in a concept tree might have properties of other concepts, functionalities (actions associated with that concept), states (Boolean expressions validating a specific state), etc. For example, the Vehicle's $IsMoving$ state holds when the vehicle speed (the $VehicleSpeed$ property) is greater than 0.

```
// e-Vehicle
CONCEPT Vehicle {
  PARENTS {eMobility.eCars.CONCEPT_TREES.Entity}
  CHILDREN {        }
  PROPS {
    PROP carDriver {
      TYPE {eMobility.eCars.CONCEPT_TREES.Driver} CARDINALITY {1} }
    PROP carPassengers {
      TYPE {eMobility.eCars.CONCEPT_TREES.Passenger} CARDINALITY {*} }
    PROP carBattery {
      TYPE {eMobility.eCars.CONCEPT_TREES.Battery} CARDINALITY {1} }
  }
  FUNCS {
    FUNC startEngine {TYPE {eMobility.eCars.CONCEPT_TREES.StartEngine}}
    FUNC stopEngine {TYPE {eMobility.eCars.CONCEPT_TREES.StopEngine}}
    FUNC accelerate {TYPE {eMobility.eCars.CONCEPT_TREES.Accelerate}}
    FUNC slowDown {TYPE {eMobility.eCars.CONCEPT_TREES.SlowDown}}
    FUNC startDriving {TYPE {eMobility.eCars.CONCEPT_TREES.StartDriving}}
    FUNC stopDriving {TYPE {eMobility.eCars.CONCEPT_TREES.StopDriving}}
  }
  STATES {
    STATE IsOperational{
NOT eMobility.eCars.CONCEPT_TREES.Vehicle.PROPS.carBattery.STATES.batteryLow }
    STATE IsMoving{ eMobility.eCars.CONCEPT_TREES.VehicleSpeed > 0 }
  }
}

CONCEPT Journey {
  PARENTS {eMobility.eCars.CONCEPT_TREES.Phenomenon}
  CHILDREN {}
  PROPS {
    PROP journeyRoute {TYPE {eMobility.eCars.CONCEPT_TREES.Route} CARDINALITY {1}}
    PROP journeyTime {TYPE {DATETIME} CARDINALITY {1}}
    PROP journeyCars {TYPE {eMobility.eCars.CONCEPT_TREES.Vehicle} CARDINALITY {*}}
  }
  STATES
  {
    STATE STATE InSufficientBattery {eMobility.eCars.CONCEPT_TREES.JourneyBatterySufficiency > 0}
    STATE InNotSufficientBattery {
      NOT eMobility.eCars.CONCEPT_TREES.Journey.STATES.InSufficientBattery}
    STATE Arrived {eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.STATES.AtEnd}
    STATE ArrivedOnTime { eMobility.eCars.CONCEPT_TREES.Journey.STATES.Arrived AND
                         (eMobility.eCars.CONCEPT_TREES.JourneyTime <=
                          eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyTime)
                        }
  }
}

CONCEPT Route {
  PARENTS {eMobility.eCars.CONCEPT_TREES.Phenomenon}
  CHILDREN {}
  PROPS {
    PROP locationA {TYPE {eMobility.eCars.CONCEPT_TREES.Location} CARDINALITY {1}}
    PROP locationB {TYPE {eMobility.eCars.CONCEPT_TREES.Location} CARDINALITY {1}}
    PROP intermediateStops {TYPE {eMobility.eCars.CONCEPT_TREES.Location} CARDINALITY {*}}
    PROP currentRoad {TYPE {eMobility.eCars.CONCEPT_TREES.Road} CARDINALITY {1}}
    PROP alternativeRoads {TYPE {eMobility.eCars.CONCEPT_TREES.Road} CARDINALITY {*}}
  }
  FUNCS {
    FUNC getCurrentLocation {TYPE {eMobility.eCars.CONCEPT_TREES.GetCurrentLocation}}
    FUNC takeAlternativeRoad {TYPE {eMobility.eCars.CONCEPT_TREES.TakeAlternativeRoad}}
    FUNC recomputeRoads {TYPE {eMobility.eCars.CONCEPT_TREES.RecomputeRoads}}
  }
  STATES {
    STATE AtBeginning {eMobility.eCars.CONCEPT_TREES.Route.FUNCS.getCurrentLocation =
                       eMobility.eCars.CONCEPT_TREES.Route.PROPS.locationA}
    STATE AtEnd {eMobility.eCars.CONCEPT_TREES.Route.FUNCS.getCurrentLocation =
                 eMobility.eCars.CONCEPT_TREES.Route.PROPS.locationB}
    STATE OnRoute { NOT eMobility.eCars.CONCEPT_TREES.Route.STATES.AtBeginning AND
                    NOT eMobility.eCars.CONCEPT_TREES.Route.STATES.AtEnd}
    STATE InHighTraffic {
      eMobility.eCars.CONCEPT_TREES.Route.PROPS.currentRoad.STATES.InHighTraffic}
    STATE InLowTraffic {
      eMobility.eCars.CONCEPT_TREES.Route.PROPS.currentRoad.STATES.InFluentTraffic}
  }
}
```

As mentioned above, the states are specified as Boolean expressions. For example, the state $Route$'s $OnRoute$ holds (is true) while the $Route$ is neither $AtBeginning$ nor at $AtEnd$ states. A concept realization is an object instantiated from that concept. As shown, a complex state might be expressed as a Boolean function over other states. For example, the $Journey$'s state $ArrivedOnTime$ is expressed as a Bollean expression involving the $Journey$'s $Arrived$ state and $Journey$'s properties.

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

3.3 Specifying Self-Adaptive Behavior

To specify self-* objectives with KnowLang, we use *goals*, *policies*, and *situations*. These are defined as explicit concepts in KnowLang, and for the eMobility Ontology we specified them under the concepts *Virtual_entity→Phenomenon→Knowledge* (see Figure 3). Figure 4, depicts a concept tree representing the specified eMobility goals. Note that most of these goals were directly interpolated from the goals model (see Figure 2).

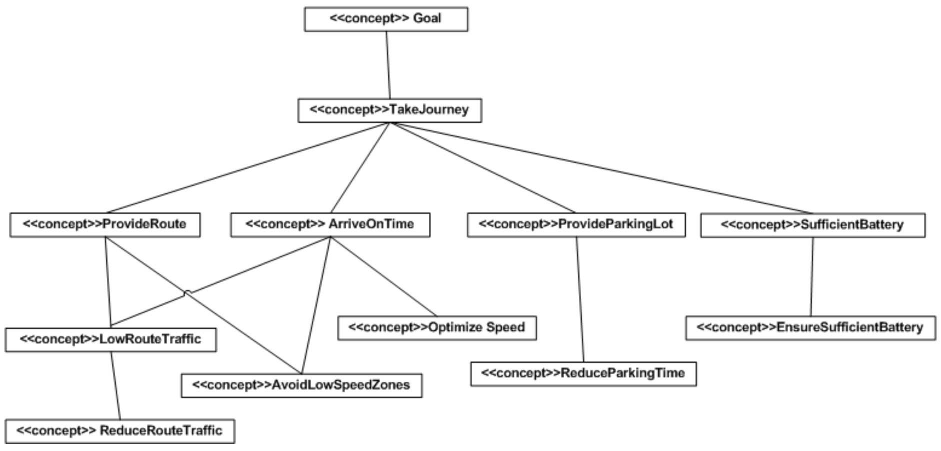

<<concept>> Goal
<<concept>>TakeJourney
<<concept>>ProvideRoute
<<concept>> ArriveOnTime
<<concept>>ProvideParkingLot
<<concept>>SufficientBattery
<<concept>>Optimize Speed
<<concept>>EnsureSufficientBattery
<<concept>>LowRouteTraffic
<<concept>>AvoidLowSpeedZones
<<concept>>ReduceParkingTime
<<concept>> ReduceRouteTraffic

Figure 4: eMobility Ontology: eMobility Goal Concept Tree

Recall that KnowLang specifies goals as functions of states where any combination of states can be involved (see Section 2.2). A goal has an arriving state (Boolean function of states) and an optional departing state (another Boolean function of states) (see Definition 5 in Section 2.2). A goal with departing state is more restrictive, i.e., it can be achieved only if the system departs from the specific goal's departing state.

The following code samples present the specification of two simple goals. Usually, goals' arriving and departing states can be either single states or sequences of states. Note that the states used to specify the goals below are specified as part of both $Journey$ and $Route$ concepts.

```
//
//==== eMobility Goals ==========================================
//
CONCEPT_GOAL ArriveOnTime {
  CHILDREN {eMobility.eCars.CONCEPT_TREES.Goal}
  PARENTS {}
  SPEC {
    DEPART { eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.STATES.AtEnd }
    ARRIVE { eMobility.eCars.CONCEPT_TREES.Journey.STATES.ArrivedOnTime }
  }
}
CONCEPT_GOAL LowRouteTraffic {
  CHILDREN {eMobility.eCars.CONCEPT_TREES.Goal}
  PARENTS {}
  SPEC {
    DEPART { eMobility.eCars.CONCEPT_TREES.Route.STATES.InHighTraffic }
    ARRIVE { eMobility.eCars.CONCEPT_TREES.Route.STATES.InLowTraffic }
  }
}
```

The following is a specification sample showing an eMobility policy called $ReduceRouteTraffic$ - as the name says, this policy is intended to reduce the route traffic. As shown, the policy is specified to handle the goal $LowRouteTraffic$ and is triggered by the situation $RouteTrafficIncreased$. Further, the policy triggers via its $MAPPING$ sections conditionally (e.g., there is a $CONDITONS$ directive that requires the $Route$'s state $OnRoute$ to be hold) the execution of a sequence of actions. When the conditions are the same, we specify a probability distribution among the $MAPPING$ sections involving same conditions (e.g., $PROBABILITY 0.7$), which represents our initial belief in action choice.

```
CONCEPT_POLICY ReduceRouteTraffic {
  CHILDREN {}
  PARENTS {eMobility.eCars.CONCEPT_TREES.Policy}
  SPEC {
    POLICY_GOAL {eMobility.eCars.CONCEPT_TREES.LowRouteTraffic}
    POLICY_SITUATIONS {eMobility.eCars.CONCEPT_TREES.RouteTrafficIncreased}
    POLICY_RELATIONS {eMobility.eCars.RELATIONS.Situation_Policy_1}
    POLICY_ACTIONS {eMobility.eCars.CONCEPT_TREES.TakeAlternativeRoad,
                    eMobility.eCars.CONCEPT_TREES.RecomputeRoads}
    POLICY_MAPPINGS {
      MAPPING {
        CONDITIONS {eMobility.eCars.CONCEPT_TREES.Route.STATES.OnRoute}
        DO_ACTIONS {eMobility.eCars.CONCEPT_TREES.Route.FUNCS.takeAlternativeRoad}
        PROBABILITY {0.7}
      }
      MAPPING {
        CONDITIONS { eMobility.eCars.CONCEPT_TREES.Route.STATES.OnRoute}
        DO_ACTIONS { eMobility.eCars.CONCEPT_TREES.Route.FUNCS.recomputeRoads,
                     eMobility.eCars.CONCEPT_TREES.Route.FUNCS.takeAlternativeRoad}
        PROBABILITY {0.3}
      }
      MAPPING {
        CONDITIONS { eMobility.eCars.CONCEPT_TREES.Route.STATES.AtBeginning}
        DO_ACTIONS { eMobility.eCars.CONCEPT_TREES.Route.FUNCS.recomputeRoads,
                     eMobility.eCars.CONCEPT_TREES.Route.FUNCS.takeAlternativeRoad}
      }
    }
  }
}
```

As specified, the probability distribution gives initial designer's preference about what actions should be executed if the system ends up in running the $ReduceRouteTraffic$ policy. Note that at runtime, the KnowLang Reasoner maintains a record of all the action executions and re-computes the probability rates every time when a policy has been applied and consecutively, actions have been executed. Thus, although initially the system will execute the function $takeAlternativeRoad$ (it has the higher probability rate of 0.7), if that policy cannot achieve its goal with this action, then the probability distribution will be shifted in favor of the function sequence $recomputeRoads, takeAlternative$-$Road$, which might be executed the next time when the system will try to apply the same policy. Therefore, probabilities are recomputed after every action execution, and thus the behavior change accordingly.

Moreover, to increase the goal-oriented autonomicity, in policy specification, we may use a special operator implemented in KnowLang called $GENERATE\_NEXT\_ACTIONS$. This operator will automatically generate the most appropriate actions to be undertaken by eMobility. The action generation is based on the computations performed by a special *reward function* implemented by the KnowLang Reasoner. The *KnowLang Reward Function* (KLRF) observes the outcome of the actions to compute the possible successor states of every possible action execution and grants the actions with special reward number considering the current system state (or states, if the current state is a composite state) and goals. KLRF is based on past experience and uses Discrete Time Markov Chains [EG05] for probability assessment after action executions [VHM$^+$12, VH15].

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

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

[1]The Bellman optimality principle: If a given state-action sequence is optimal, and we were to remove the first state and

sequence of actions.

As mentioned above, policies are triggered by situations. Therefore, while specifying policies handling eMobility objectives, we need to think of important situations that may trigger those policies. These situations shall be eventually outlined by scenarios. A single policy requires to be associated with (related to) at least one situation (see Section 2.2), but for polices handling self-* objectives we eventually need more situations. Actually, because the policy-situation relation is bidirectional, it is maybe more accurate to say that a single situation may need more policies, those providing alternative behaviors or execution paths out of that situation. The following code represents the specification of two situations: $RouteTrafficIncreased$ and $BatteryIsInsufficient$ where both were used for the specification of policies. For example, the $RouteTrafficIncreased$ situation was used to specify the $ReduceRouteTraffic$ policy.

```
  CONCEPT_SITUATION RouteTrafficIncreased {
    CHILDREN {}
    PARENTS {eMobility.eCars.CONCEPT_TREES.Situation}
    SPEC {
      SITUATION_STATES {eMobility.eCars.CONCEPT_TREES.Route.STATES.InHighTraffic}
      SITUATION_ACTIONS {eMobility.eCars.CONCEPT_TREES.TakeAlternativeRoad}
    }
  }
  CONCEPT_SITUATION BatteryIsInsufficient { // battery is insufficient to complete the journey
    CHILDREN {}
    PARENTS {eMobility.eCars.CONCEPT_TREES.Situation}
    SPEC {
      SITUATION_STATES {eMobility.eCars.CONCEPT_TREES.Journey.STATES.InNotSufficientBattery}
      SITUATION_ACTIONS {eMobility.eCars.CONCEPT_TREES.FindNearestChargeStation,
                         eMobility.eCars.CONCEPT_TREES.GoToChargeStation,
                         eMobility.eCars.CONCEPT_TREES.ChargeBattery,
                         eMobility.eCars.CONCEPT_TREES.ReplaceVehicle}
    }
  }
```

As shown, the situation is specified with $SITATION\_STATES$ (e.g., $InHighTraffic$) and $SITUATION\_ACTIONS$ (e.g., $TakeAlterna\text{-}tiveRoad$). To consider a situation effective (i.e., the system is currently in that situation), the situation states must be respectively effective (evaluated as true). For example, the situation $RouteTraf\text{-}ficIncreased$ is effective if the $Route$'s state $InHighTraffic$ is effective (is hold). The possible actions define what actions can be undertaken once the system falls in a particular situation. For example, the $RouteTrafficIncreased$ situation has one possible action: $TakeAlternativeRoad$.

Recall that situations are related to policies via relations (see Definition 2 in Section 2.2). The following code demonstrates how we related the situation $RouteTrafficIncreased$ to the policy $Reduce\text{-}RouteTraffic$.

```
RELATION Situation_Policy_1{
  RELATION_PAIR {
    eMobility.eCars.CONCEPT_TREES.RouteTrafficIncreased,
    eMobility.eCars.CONCEPT_TREES.ReduceRouteTraffic}
  }
}
```

In general, a self-adaptive system has sensors that connect it to the world and eventually help it listen to its internal components. These sensors generate raw data that represent the physical characteristics of the world. The representation of monitoring sensors in KnowLang is handled via the explicit *Metric concept* [VHM$^+$12, VH15]. In our approach, we assume that eMobility sensors are controlled by software drivers (e.g., implemented in C++) where appropriate methods are used to control a sensor and read data from it. By specifying a $Metric$ concept we introduce a class of sensors to the KB and by specifying objects, instances of that class, we represent the real sensor. KnowLang allows the specification of four different types of metrics [VHM$^+$12, VH15]:

- *RESOURCE* - measure resources like capacity;

action, the remaining sequence is also optimal (with the second state of the original sequence now acting as initial state).

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

The following is a specification of metrics mainly used to assist the specification of states in the specification of the eMobility concepts (see Section 3.2).

```
// metrics
CONCEPT_METRIC RoadTrafficLevel {
  CHILDREN {}
  PARENTS {eMobility.eCars.CONCEPT_TREES.Metric}
  SPEC {
    METRIC_TYPE { ENVIRONMENT }
    METRIC_SOURCE {        "ECarClass.GetRoadTrafficLevel" }
    DATA_TYPE { NUMBER }
  }
}
CONCEPT_METRIC BatteryEnergyLevel {
  CHILDREN {}
  PARENTS {eMobility.eCars.CONCEPT_TREES.Metric}
  SPEC {
    METRIC_TYPE { RESOURCE }
    METRIC_SOURCE {        "ECarClass.GetBatteryEnergyLevel" }
    DATA_TYPE { NUMBER }
  }
}
CONCEPT_METRIC JourneyBatterySufficiency {
  CHILDREN {}
  PARENTS {eMobility.eCars.CONCEPT_TREES.Metric}
  SPEC {
    METRIC_TYPE { RESOURCE }
    METRIC_SOURCE {        "ECarClass.GetJourneyBatterySufficiency" }
    DATA_TYPE { BOOLEAN }
  }
}
CONCEPT_METRIC VehicleSpeed {
  CHILDREN {}
  PARENTS {eMobility.eCars.CONCEPT_TREES.Metric}
  SPEC {
    METRIC_TYPE { RESOURCE }
    METRIC_SOURCE {        "ECarClass.GetVehicleSpeed" }
    DATA_TYPE { NUMBER }
  }
}
CONCEPT_METRIC JourneyTime {
  CHILDREN {}
  PARENTS {eMobility.eCars.CONCEPT_TREES.Metric}
  SPEC {
    METRIC_TYPE { RESOURCE }
    METRIC_SOURCE {        "ECarClass.GetJourneyTime" }
    DATA_TYPE { DATETIME }
  }
}
```

4 Formalizing Swarm Robotics with KnowLang

Aside from complex mechanics and electronics, building robots is about the challenge of interacting with a dynamic and unpredictable world, which requires the presence of intelligence. In swarm robotics systems, in addition to this challenge, we also need to deal with the dynamic local interactions among robots, often resulting in emergent behavior at the level of the entire swarm. Real swarm intelligence systems such as social insects, bird flocks and fish schools, leverage such parallelism to achieve remarkable efficiency and robustness to hazards. The prospect of replicating the performance of natural systems and their incredible ability of self-adaptation is the main motivation in the study of swarm robotics systems.

Swarm robotics brings most of the challenges that the theories and methodologies developed for self-adaptive systems are attempting to solve. Hence, self-adaptation has emerged as an important paradigm making a swarm robotics system capable of modifying the system behavior and/or structure in response to increasing workload demands and changes in the operational environment. Note that robotic artificial intelligence (AI) mainly excels at formal logic, which allows it, for example, to find the appropriate action from hundreds of possible actions.

The Ensemble of Robots Case Study targets swarms of intelligent robots with self-awareness capabilities that help the entire swarm acquire the capacity to reason, plan, and autonomously act [SRA+11, SMP+12]. The case study relies on the marXbot robotics platform [BBR+11], which is a modular research robot equipped with a set of devices that help the robot interact with other robots of the swarm or the robotic environment. The environment is defined as an arena where special cuboid-shaped obstacles are present in arbitrary positions and orientations. Moreover, the environment may contain a number of light sources, usually placed behind the goal area, which act as environmental cues used as shared reference frames among all robots.

Each marXbot robot is equipped with a set of devices to interact with the environment and with other robots of the swarm:

- a light sensor, that is able to perceive a noisy light gradient around the robot in the 2D plane;
- a distance scanner that is used to obtain noisy distances and angular values from the robot to other objects in the environment;
- a range and bearing communication system, with which a robot can communicate with other robots that are in line-of-sight;
- a gripper, that is used to physically connect to the transported object;
- two wheels independently controlled to set the speed of the robot.

Currently, the marXbots robots are able to work in teams where they coordinate based on simple interactions in group tasks. For example, a group of marXbots robots may collectively move a relatively heavy object from point A to point B by using their grippers.

For the purpose of the Ensemble of Robots case study, we developed a simple scenario that requires self-adaptive behavior of the individual marXbot robots [SHP+13]. In this scenario, a team of marXbot robots, called *rescuers*, is deployed in a special area, called a *deployment area*. We imagine that some kind of disaster has happened, and the environment is occasionally obstructed by *debris* that the robots can move around. In addition, a portion of the environment is dangerous for robot navigation due to the presence of *radiation*. We assume that prolonged exposition to radiation damages the robots. For example, short-term exposition increases a robot's sensory noise. Long-term damage, eventually, disables the robot completely. To avoid damage, the robots can use debris to build a *protective wall*.

Further, we imagine that a number of *victims* are trapped in the environment and must be rescued by the robots. Each victim is suffering a different injury. The robots must calculate a suitable *rescuing behavior* that maximizes the number of victims rescued. A victim is considered rescued when it is deposited in the deployment area alive. To perform its activities, a robot must take into account that it has limited energy.

4.1 ARE for Swarm Robotics

Following the scenario described above, we applied the ARE approach (see Section 2.1) and derived the goals along with the *self-* objectives* assisting these goals when self-adaptation is required. Further, based on the rationale above, we applied the ARE approach and derived the system's goals along with the self-* objectives assisting these goals when self-adaptation is required.

Figure 5 depicts the ARE goals model for swarm robotics where goals are organized hierarchically at three different levels. As shown, the goals from the first two levels (e.g., "Rescue Victims", "Protect

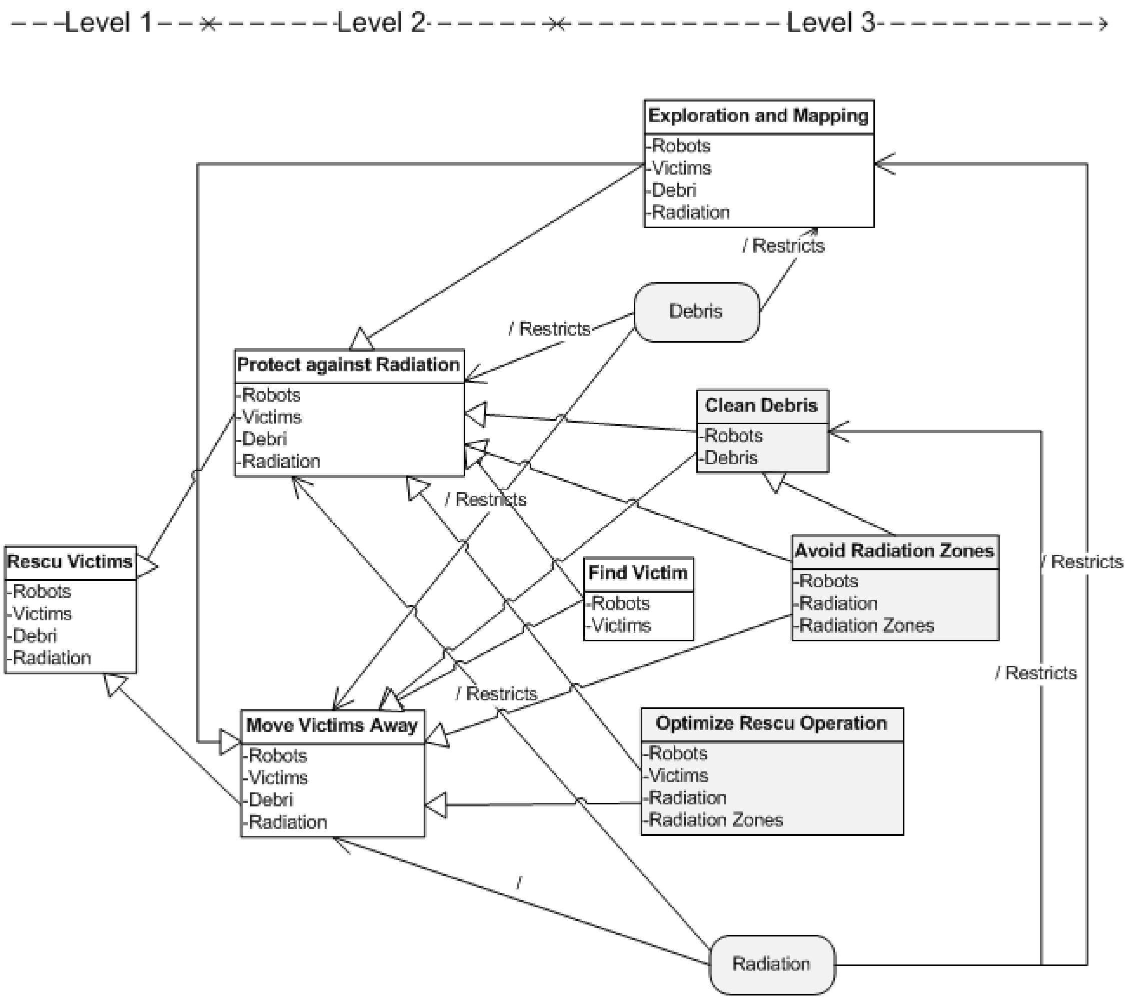


Figure 5: Swarm Robotics Goals Model with Self-* Objectives

against Radiation", and "Move Victims away") are *main system goals* captured at different levels of abstraction. The 3rd level is resided by *self-* objectives* (e.g., "Clean Debris", "Optimize Rescue Operation", and "Avoid Radiation Zones") and *supportive goals* (e.g., "Exploration and Mapping" and "Find Victim") associated with and assisting the 2nd-level goals. Basically, all self-* objectives inherit the system goals they assist by providing behavior alternatives with respect to these system goals. The system switches to one of the assisting self-* objectives when alternative autonomous behavior is required (e.g., a robot needs to avoid a radiation zone). In addition, Figure 5 depicts some of the environmental constraints (e.g., "Radiation" and "Debris"), which may cause self-adaptation.

4.2 Specifying Swarm Robotics Ontology

In order to specify the autonomy requirements for swarm robotics, the first step is to specify a knowledge base (KB) representing the swarm robotics system in question, i.e., robots, victims, radiation, debris, etc. To do so, we need to specify ontology structuring the knowledge domain of the case study. Note that this domain is described via domain-relevant concepts and objects (concept instances) related through relations. To handle explicit concepts like situations, goals, and policies, we grant some of the domain concepts with explicit state expressions where a state expression is a Boolean expression over the ontology (see Definition 6 in Section 2.2).

Figure 6, depicts a graphical representation of the swarm robotics ontology relating most of the domain concepts within a swarm robotics system. Note that the relationships within a concept tree are "is-a" (inheritance), e.g., the $Radiation\_Zone$ concept is an $EnvironmentEntity$ and the $Action$ concept is a $Knowledge$ and consecutively $Phenomenon$, etc. Most of the concepts presented in Figure 6 were derived from the Swarm Robotics Goals Model (see Figure 5). Other concepts are considered *explicit* and were derived from the KnowLang specification model [VHM+12, VH15].

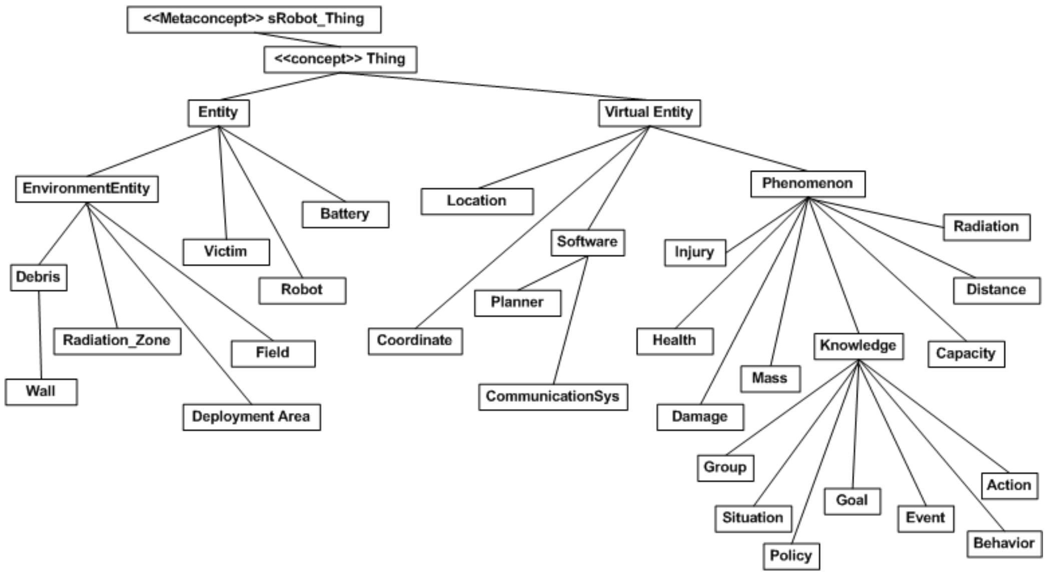


Figure 6: Swarm Robotics Ontology Specified with KnowLang

The following is a sample of the KnowLang specification representing the *Robot* concept. As specified, the concept has properties of other concepts, functionalities (actions associated with that concept), states (Boolean expressions validating a specific state), etc. For example, the $IsOperational$ state holds when the robot's battery (the $rBattery$ property) is not in the $batteryLow$ state and the robot itself is not in the $IsDamaged$ state.

```
CONCEPT Robot { ....
 PROPS {
  PROP rBattery {TYPE{swarmRobots.robots.CONCEPT_TREES.Battery} CARDINALITY{1}}
  PROP rPlanner {TYPE{swarmRobots.robots.CONCEPT_TREES.Planner} CARDINALITY{1}}
  PROP rCommunicationSys {TYPE{swarmRobots.robots.CONCEPT_TREES.CommunicationSys} CARDINALITY{1}}
  PROP liftCapacity {TYPE{NUMBER} CARDINALITY{1}}
  PROP dragCapacity {TYPE{swarmRobots.robots.CONCEPT_TREES.Capacity} CARDINALITY{1}}
  PROP rDamages {TYPE{swarmRobots.robots.CONCEPT_TREES.Damage} CARDINALITY{*}}
  PROP distDebries {TYPE{swarmRobots.robots.CONCEPT_TREES.Dstance_to_Debries} CARDINALITY{1}}
  PROP victimToCareOf {TYPE{swarmRobots.robots.CONCEPT_TREES.Victim} CARDINALITY{1}}}
 FUNCS {
  FUNC plan {TYPE {swarmRobots.robots.CONCEPT_TREES.Plan}}
  FUNC explore {TYPE {swarmRobots.robots.CONCEPT_TREES.Explore}}
  FUNC selfCheck {TYPE {swarmRobots.robots.CONCEPT_TREES.CheckForDamages}}
  FUNC dragVictimAway {TYPE {swarmRobots.robots.CONCEPT_TREES.DragVictim}}
  FUNC carryVictim {TYPE {swarmRobots.robots.CONCEPT_TREES.CarryVictim}}
  FUNC buildWall {TYPE {swarmRobots.robots.CONCEPT_TREES.BuildWall}}}
 STATES {
  STATE IsOperational{ NOT swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.rBattery.STATES.batteryLow AND
                       NOT swarmRobots.robots.CONCEPT_TREES.Robot.STATES.IsDamaged }
  STATE IsDamaged { swarmRobots.robots.CONCEPT_TREES.Robot.FUNCS.selfCheck > 0 }
  STATE IsPlaning { IS_PERFORMING{swarmRobots.robots.CONCEPT_TREES.Robot.FUNCS.plan} }
  STATE IsExploring { IS_PERFORMING{swarmRobots.robots.CONCEPT_TREES.Robot.FUNCS.explore} }
  STATE HasDebrisNearby { swarmRobots.robots.CONCEPT_TREES.Victim.PROPS.distDeplArea < 3 } //less than 3 m
}}
```

As we have already noticed, the *states* are extremely important to the specification of *goals*, *situations*, and *policies*. For example, states help the KnowLang Reasoner determine at runtime whether the system is in a particular situation or a particular goal has been achieved. The following code sample presents a partial specification of a simple goal.

```
CONCEPT_GOAL Protect_Victim_against_Radiation { ....
 SPEC {
  DEPART { swarmRobots.robots.CONCEPT_TREES.Victim.STATES.underRadiation  }
  ARRIVE { swarmRobots.robots.CONCEPT_TREES.Victim.STATES.radiationSafe }}}
```

4.3 Specifying Self-Adaptive Behavior

The following is the specification of a policy called $ProtectVictimAgainstRadiation$. As shown, the policy is specified to handle the $Protect\_Victim\_against\_Radiation$ goal and is triggered by the situation $VictimNeedsHelp$. Further, the policy triggers via its $MAPPING$ sections conditionally the execution of a sequence of actions. When the conditions are the same, we specify a probability distribution among the $MAPPING$ sections involving same conditions (e.g., $PROBABILITY\{0.6\}$), which represents our initial belief in action choice.

```
CONCEPT_POLICY ProtectVictimAgainstRadiation { ....
 SPEC {
  POLICY_GOAL { swarmRobots.robots.CONCEPT_TREES.Protect_Victim_against_Radiation }
  POLICY_SITUATIONS { swarmRobots.robots.CONCEPT_TREES.VictimNeedsHelp }
  POLICY_RELATIONS { swarmRobots.robots.RELATIONS.Policy_Situation_1 }
  POLICY_ACTIONS { swarmRobots.robots.CONCEPT_TREES.DragVictim,
   swarmRobots.robots.CONCEPT_TREES.CarryVictim,swarmRobots.robots.CONCEPT_TREES.BuildWall}
  POLICY_MAPPINGS {
   MAPPING {
    CONDITIONS { swarmRobots.robots.CONCEPT_TREES.Robot.STATES.IsOperational AND
     swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.victimToCareOf.PROPS.victimMass >
     swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.liftCapacity AND
     swarmRobots.robots.CONCEPT_TREES.Robot.STATES.HasDebrisNearby}
    DO_ACTIONS {swarmRobots.robots.CONCEPT_TREES.Robot.FUNCS.dragVictimAway} PROBABILITY {0.6}}
   MAPPING {
    CONDITIONS { swarmRobots.robots.CONCEPT_TREES.Robot.STATES.IsOperational AND
     swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.victimToCareOf.PROPS.victimMass >
     swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.liftCapacity AND
     swarmRobots.robots.CONCEPT_TREES.Robot.STATES.HasDebrisNearby}
    DO_ACTIONS {swarmRobots.robots.CONCEPT_TREES.Robot.FUNCS.buildWall} PROBABILITY {0.4}}
   MAPPING {
    CONDITIONS { swarmRobots.robots.CONCEPT_TREES.Robot.STATES.IsOperational AND
     swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.victimToCareOf.PROPS.victimMass <=
     swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.liftCapacity}
    DO_ACTIONS {swarmRobots.robots.CONCEPT_TREES.Robot.FUNCS.carryVictim} PROBABILITY {0.6}}
   MAPPING {
    CONDITIONS { swarmRobots.robots.CONCEPT_TREES.Robot.STATES.IsOperational AND
     swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.victimToCareOf.PROPS.victimMass <=
     swarmRobots.robots.CONCEPT_TREES.Robot.PROPS.liftCapacity}
    DO_ACTIONS { swarmRobots.robots.CONCEPT_TREES.Robot.FUNCS.dragVictimAway} PROBABILITY {0.4}
}}}}
```

As specified, the probability distribution gives the designer's initial preference about what actions should be executed if the system ended up running that policy. Note that at runtime, the KnowLang Reasoner maintains a record of all the action executions and re-computes the probability rates every time when a policy has been applied and subsequently, actions have been executed. Thus, although initially the system will execute the function $dragVictimAway$ (it has the higher probability rate of 0.6), if that policy cannot achieve its goal with this action, then the probability distribution will be shifted in favor of the function $buildWall$, which may be executed the next time when the system will try to apply the same policy. Therefore, probabilities are recomputed after every action execution, and thus the behavior changes accordingly.

5 KnowLang Reasoner

A very challenging task is the R&D of the inference mechanism providing for *knowledge reasoning and awareness*. In order to support reasoning about self-adaptive behavior and to provide a KR gateway for communication with the KB, we have developed the KnowLang Reasoner. The reasoner communicates with the system and operates in the KR Context, a context formed by the represented knowledge (see Figure 7).

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

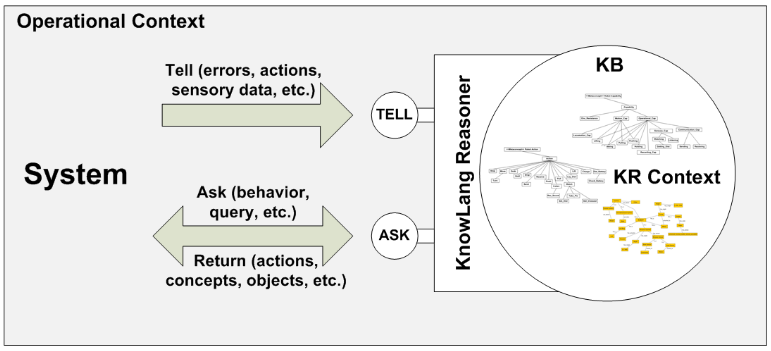

Operational Context
System
Tell (errors, actions, sensory data, etc.)
Ask (behavior, query, etc.)
Return (actions, concepts, objects, etc.)
TELL
ASK
KnowLang Reasoner
KB
KR Context

Figure 7: KnowLang Reasoner

the KR Context and on the KR symbols (represented knowledge). The system talks to the reasoner via special ASK and TELL Operators allowing for knowledge queries and knowledge updates (see Figure 7). Upon demand, the KnowLang Reasoner can also build up and return a self-adaptive behavior model - a chain of actions to be realized in the environment or in the system.

5.1 ASK and TELL Operators

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

So far, we have developed the operational semantics of the following TELL and ASK Operators [Vas12]:

- $TELL\_ERR$ - 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;
- $TELL\_BEHAVIOR$ (behavior) - tells about overall behavior performance, i.e., whether all the actions outlining a behavior have succeeded;
- $TELL\_OBJ\_UPDATE$ - 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.

5.2 The ASK_BEHAVIOR Operator

This subsection provides a brief presentation of the *operational semantics* of the $ASK\_BEHAVIOR$ KB Operator [Vas12]. For more information on the operational semantics of the other KnowLang KB Operators, please consult [Vas12].

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 - σ states for Operational Context (OC) and σ' states for Knowledge Representation Context (KRC) (see Figure 7). For clarity reasons, we do not show the change in KRC after updates have been made in that context.

$$(1)\,\frac{\sigma\xrightarrow{ask\_behavior()}\sigma'}{\langle ASK\_BEHAVIOR,\sigma'\rangle\longrightarrow\langle findCurrentSituation(),\sigma'\rangle}$$

$$(2)\,\frac{\sigma\xrightarrow{ask\_behavior()}\sigma'\,\langle findCurrentSituation(),\sigma'\rangle\longrightarrow\langle si,\sigma'\rangle}{\langle findSitnPolcyRltns(si),\sigma'\rangle\longrightarrow\langle R_{si},\sigma'\rangle}$$

$$(3)\,\frac{\sigma\xrightarrow{ask\_behavior()}\sigma'\,\langle findSitnPolcyRltns(si),\sigma'\rangle\longrightarrow\langle R_{si},\sigma'\rangle}{\langle max(R_{si}),\sigma'\rangle\longrightarrow\langle \pi_{si},\sigma'\rangle}$$

$$(4)\ \langle \pi,\sigma'\rangle \longrightarrow \langle applyPolicy(\pi),\sigma'\rangle$$

$$(5)\,\frac{\begin{array}{c}\langle \pi_{si},\sigma'\rangle\longrightarrow\langle applyPolicy(\pi_{si}),\sigma'\rangle\\ \forall n_\pi\in N_\pi\bullet\langle n_\pi,\sigma'\rangle\longrightarrow\langle TRUE,\sigma'\rangle\end{array}}{\langle map(\pi_{si},N_\pi,A_\pi,Z),\sigma'\rangle\longrightarrow\langle <A'_\pi,Z'>,\sigma'\rangle}\ A'_\pi\subseteq A_\pi$$

$$(6)\,\frac{\begin{array}{c}\langle \pi_{si},\sigma'\rangle\longrightarrow\langle applyPolicy(\pi_{si}),\sigma'\rangle\\ \langle map(\pi_{si},N_\pi,A_\pi,Z),\sigma'\rangle\longrightarrow\langle <A'_\pi,Z'>,\sigma'\rangle\\ \langle max(Z'),\sigma'\rangle\longrightarrow\langle z,\sigma'\rangle\end{array}}{\langle getProbableActions(<A'_\pi,Z'>,z),\sigma'\rangle\longrightarrow\langle <A''_\pi,z>,\sigma'\rangle}$$

$$(7)\,\frac{\begin{array}{c}\langle \pi_{si},\sigma'\rangle\longrightarrow\langle applyPolicy(\pi_{si}),\sigma'\rangle\\ \langle map(\pi_{si},N_\pi,A_\pi,Z),\sigma'\rangle\longrightarrow\langle <A'_\pi,Z'>,\sigma'\rangle\\ \langle getProbableActions(<A'_\pi,Z'>,z),\sigma'\rangle\longrightarrow\langle <A''_\pi,z>,\sigma'\rangle\end{array}}{\langle recordBehavior(\pi_{si},A''_\pi),\sigma'\rangle\longrightarrow\left\langle b^{\pi}_{si},\sigma'\right\rangle}$$

$$(8)\,\frac{\sigma\xrightarrow{ask\_behavior()}\sigma'\,\langle recordBehavior(\pi_{si},A_\pi),\sigma'\rangle\longrightarrow\left\langle b^{\pi}_{si},\sigma'\right\rangle}{\sigma'\xrightarrow{return(b^{\pi}_{si})}\sigma}$$

As shown in Rule 1, 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 operating in the KRC. 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 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 2), 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 2). Next, the relation with the *highest probability rate* is selected (recall that KnowLang Relations may be associated with a probability rate - see Definition 2 in Section 2.2), which helps to determine the *most appropriate policy* for that particular situation (see the conclusion in Rule 3). The selected policy is applied (see Rule 4). The evaluation of a 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 4 in Section 2.2). This operation selects *pairs "actions subset"-"probability rate"* (see the conclusion in Rule 5). 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 6). The extracted subset of possible actions has to be recorded as a *behavior model* (see the conclusion in Rule 7 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 with a context switching back to OC σ (see Rule 8). Note that the behavior model must comprise only actions allowed to be executed from the actual situation (see Definition 7 in Section 2.2).

5.3 Reasoner Implementation

The KnowLang Reasoner is a comprehensive reasoning engine that operates on the KnwoLang specifications to derive self-adaptive behavior. In addition, via the predefined ASK and TELL operators, it provides KB querying and KB updating mechanisms (see Figure 7).

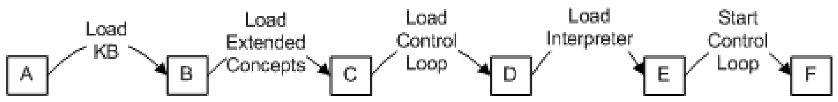


Figure 8: KnowLang Reasoner Startup Process

Therefore, in order to function, the reasoner requires as initial resource an already compiled, yet KnowLang-specified KB. As shown in Figure 8 there are a few important operations performed by the reasoner at startup. Along with loading the compiled KB into a tree of optimized KnowLang tokens, these operations also build special information-retrieval and information-update structures and load the implementation of multiple search algorithms. Finally, the startup process loads the *awareness control loop* [VHBM13] and a special query interpreter, and starts the control loop.

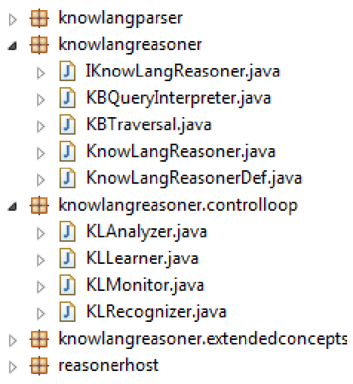

knowlangparser
knowlangreasoner
IKnowLangReasoner.java
KBQueryInterpreter.java
KBTraversal.java
KnowLangReasoner.java
KnowLangReasonerDef.java
knowlangreasoner.controlloop
KLAnalyzer.java
KLLearner.java
KLMonitor.java
KLRecognizer.java
knowlangreasoner.extendedconcepts
reasonerhost

Figure 9: KnowLang Reasoner Java Packages

Similar to the KnowLang Framework, the KnowLang Reasoner has been implemented in Java. At the time of this document writing, the KnowLang Reasoner's implementation contained approximately 7000 lines of Java code, organized in four Java packages (see Figure 9). These java packages host all the necessary classes needed at both startup and runtime.

5.3.1 KnowLang Reasoner Classes

Figure 10 presents the major classes implementing the KnowLang Reasoner. As shown, there are three token classes used by the reasoner to load the KnowLang-compiled KB - *KnowLangToken*, *KnowLangTierToken*, and *KnowLangCodeToken*. These classes are identical to their counterparts used by the KnowLang Framework [VHBM13] and the reasoner uses them to load a compiled KB into a special *declarative tree*. The *KnowLangToken* class comprises everything needed to present the words of a KnowLang specification. The *KnowLangTierToken* class is an extension of *KnowLangToken* and is used by the reasoner to reconstruct from the KB the previously-specified concepts with their states, properties and functionalities, along with all the objects and relations. In addition, the *KnowLangTierToken* class contains a reference to the *KnowLangCodeToken* class, which is basically used to store some extra code generated by the KnowLang Framework and stored in the compiled KB.

If we consider the startup process shown in Figure 8, in stage "B", the reasoner has the compiled KB loaded into the *KB declarative tree*. The *KnowLangReasonerDef* class defines the declarative tree as following:

```
public static Vector<KnowLangTierToken> vsDeclarationTree = new Vector<KnowLangTierToken>();
```

In addition to the declarative tree, that class also defines vectors of special classes intended to optimize the reasoning process. These classes are *ExtendedKLState*, *ExtendedKLMetric*, *ExtendedKLGoal*, *ExtendedKLEvent*, *ExtendedKLAction*, *ExtendedKLSituation*, *ExtendedKLPolicy*, *ExtendedKLRelation*, and *ExtendedKLGroup*. These classes provide built-in functionality to facilitate the reasoning about the KnowLang's explicit concepts [VHBM13]. Basically, each one of these classes encapsulates an instance of KnowLangTierToken representing the specification of an explicit concept. For example, the *ExtendedKLState* class encapsulates a token representing the specification of a KnowLang-specified state and to optimize the reasoner, provides methods that operate over that state, e.g., the *ExtendedKLState* class implements a mechanism for state evaluation (recall that states in KnowLang are specified and evaluated as Boolean expressions).

Here, if we consider the startup process shown in Figure 8, in stage "C", the reasoner has loaded all the extended concepts, i.e., it has created vectors of *ExtendedKL** classes. Note that extensive search algorithms operating over these vectors and the declarative tree are implemented by the *KBTraversal* class. This class is heavily used by the reasoner to find and refine needed concepts, properties, functions, etc. The following is an implementation of such an algorithm, intended to discover all the goals in the KB.

```
private Vector<KnowLangTierToken> findAllGoals(Vector<KnowLangTierToken> pvConceptTree)
{
  Vector<KnowLangTierToken> rvResult = new Vector<KnowLangTierToken>();
  Enumeration<KnowLangTierToken> tokens = pvConceptTree.elements();
  KnowLangTierToken tierToken = null;
  while (tokens.hasMoreElements())
  {
    tierToken = tokens.nextElement();
    if ( tierToken.getTierName().equals(KnowLangReasonerDef.TIER_GOAL) )
      rvResult.add(tierToken);
    else
      rvResult.addAll(findAllGoals(tierToken.getTokenVector()));
  }
  return rvResult;
}
```

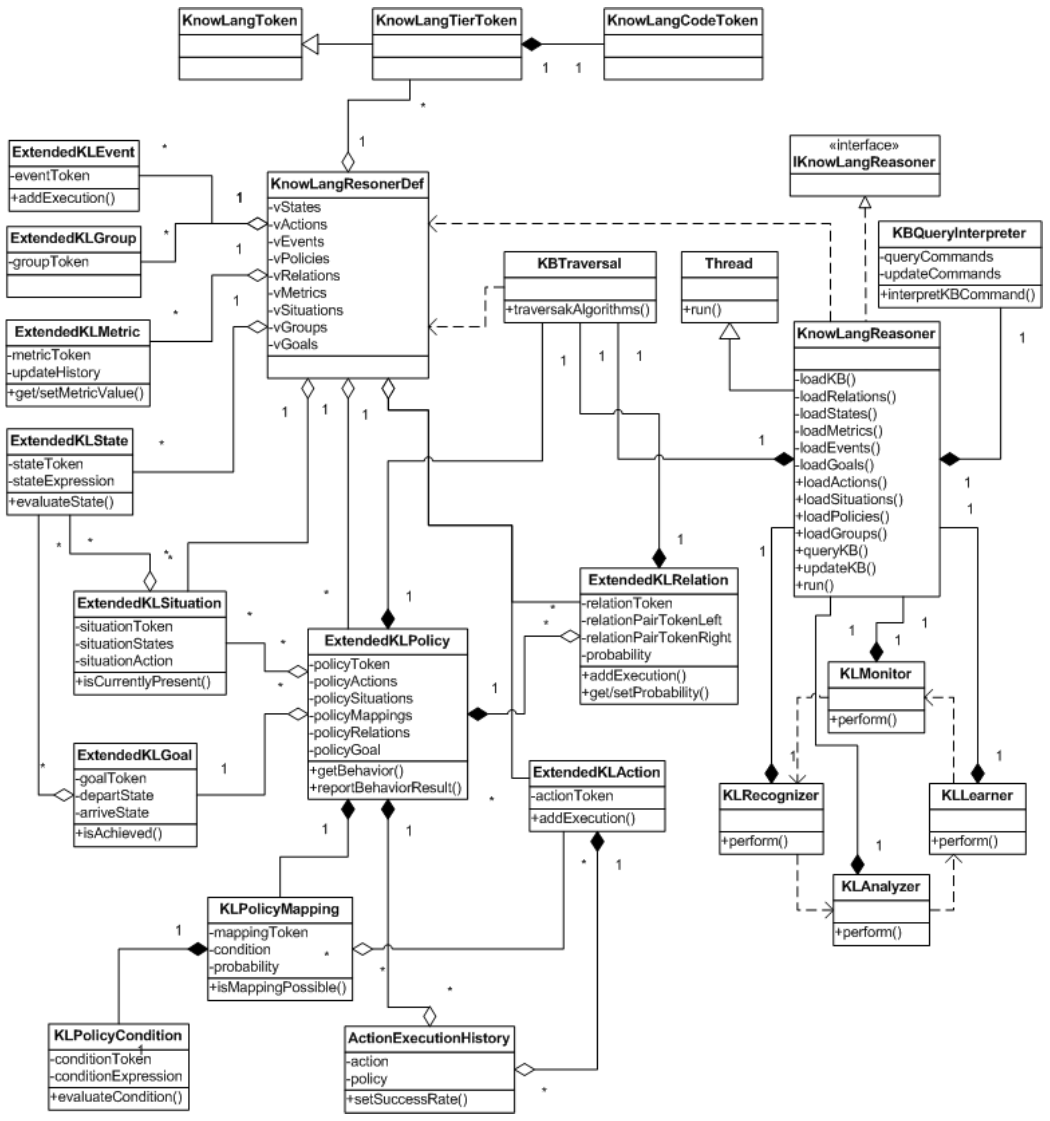

KnowLangToken
KnowLangTierToken
KnowLangCodeToken
«interface» IKnowLangReasoner
ExtendedKLEvent -eventToken +addExecution()
ExtendedKLGroup -groupToken
KnowLangResonerDef -vStates -vActions -vEvents -vPolicies -vRelations -vMetrics -vSituations -vGroups -vGoals
KBTraversal +traversakAlgorithms()
Thread +run()
KBQueryInterpreter -queryCommands -updateCommands +interpretKBCommand()
ExtendedKLMetric -metricToken -updateHistory +get/setMetricValue()
KnowLangReasoner -loadKB() -loadRelations() -loadStates() -loadMetrics() -loadEvents() -loadGoals() +loadActions() +loadSituations() +loadPolicies() +loadGroups() +queryKB() +updateKB() +run()
ExtendedKLState -stateToken -stateExpression +evaluateState()
ExtendedKLRelation -relationToken -relationPairTokenLeft -relationPairTokenRight -probability +addExecution() +get/setProbability()
ExtendedKLSituation -situationToken -situationStates -situationAction +isCurrentlyPresent()
ExtendedKLPolicy -policyToken -policyActions -policySituations -policyMappings -policyRelations -policyGoal +getBehavior() +reportBehaviorResult()
KLMonitor +perform()
ExtendedKLGoal -goalToken -departState -arriveState +isAchieved()
ExtendedKLAction -actionToken +addExecution()
KLRecognizer +perform()
KLLearner +perform()
KLAnalyzer +perform()
KLPolicyMapping -mappingToken -condition -probability +isMappingPossible()
KLPolicyCondition -conditionToken -conditionExpression +evaluateCondition()
ActionExecutionHistory -action -policy +setSuccessRate()

Figure 10: KnowLang Reasoner Class Diagram

The *KnowLangReasoner* class implements the control functionality of the KnowLang Reasoner. As shown in the class diagram, this class performs the startup process (see Figure 8) and controls the execution of a built-in control loop, along with handling all the queries addressed to the reasoner by the host application. Recall that the host application runs the reasoner and talks to it via ASK and $TELL$ operators (see Figure 7). These operators are exposed to the host application via the *IKnowLangReasoner* interface, which is implemented by the *KnowLangReasoner* class (see Figure 10). In addition, the ASK and $TELL$ operators are supported by definitions and methods provided by the *KBQueryInterpreter* class. Basically, this class interprets query and update commands provided to the KnowLang Reasoner by human users, e.g., via a command line. For example, the *KBQueryInterpreter* class interprets commands like "*ask behavior*" and "*tell sensor_val*", which are interpreted as calls of the corresponding ASK and $TELL$ operators.

The *KnowLangReasoner* class also creates and controls the *awareness control loop*, which is implemented by the four inter-connected classes *KLMonitor*, *KLRecognizer*, *KLAnalyzer*, and *KLLearner*. Each one of these classes implements a special method called *perform()*, which is called by the reasoner in any iteration of an endless loop run by the reasoner's *run()* method (the *KnowLangReasoner* class extends the predefined *Thread* class). For example, part of the KLAnalyzer's *perform()* method evaluates all the states specified in the KB as following:

```
Enumeration<ExtendedKLState> eStates = KnowLangReasonerDef.vsStates.elements();
while (eStates.hasMoreElements())
{
  eStates.nextElement().evaluateState();
}
```

In that way, the reasoner constantly evaluates all the states and deducts new state changes, thus emerging as awareness about new situations, and realization of goals (recall that states are used to express situations and goals).

5.3.2 A Closer Look at the Reasoner's Implementation

This section looks inside of the implementation of some of the reasoner's functionality as demonstrative examples. The first example presents how the reasoner does the mapping selection, when a policy has been chosen to handle a specific situation (for the relation between policies and situations see Section 2.2). The method below, called *getMappingActions()* is implemented by the *ExtendedKLPolicy* class to compute what *policy mapping* is the most appropriate one at the time of policy running, to extract the actions of the selected mapping. The method implements the *probability distribution condition* at the level of policy mappings, which helps the reasoner decide what sequence of actions to realize when multiple mappings are feasible.

```
private Vector<ExtendedKLAction> getMappingActions()
{
  KLPolicyMapping theMaping = null;
  Vector<ExtendedKLAction> vActions = new Vector<ExtendedKLAction>();
  Enumeration<KLPolicyMapping> ePolicyMappings = policyMappings.elements();
  while ( ePolicyMappings.hasMoreElements() )
  {
    KLPolicyMapping policyMapping = ePolicyMappings.nextElement();
    //Considers probability distribution among the mappings
    if ( policyMapping.isMappingPossible() )
    {
      if ( theMaping == null ) theMaping = policyMapping;
      else if (policyMapping.getProbabilityDouble() > theMaping.getProbabilityDouble())
        theMaping = policyMapping;
    }
  }
}
```

Here, as implemented after extracting all the policy's mappings, the method finds among the possible (or feasible) mappings the one with the highest probability rate.

The second example demonstrates how the reasoner performs a Boolean evaluation of KnowLang-specified states. To provide this reasoning capability, the *ExtendedKLState* class implements a method called *evaluationState()* (see Figure 10). In that way, every state loaded by the reasoner can self-evaluate itself on demand. Note that all the concepts specified in KnowLang have an intrinsic *Value* property that holds *string* values. This *Value* property is used when evaluating Boolean expressions involving concepts or objects (concepts' instances). When evaluating the *Value* properties of concepts, as part of the evaluation of Boolean expressions, they are interpreted as *numbers*, Boolean values (*true* or *false*), or *strings*. In such evaluations, the correctness of the Boolean expressions is handled in a fault-tolerant fashion, i.e., if an expression cannot be evaluated due to an error, it is assigned a *false* value.

```
public Boolean evaluateState()
{
  String sExpression = "";
  Enumeration<KnowLangTierToken> exprEntts = vStateExpression.elements();
  while (exprEntts.hasMoreElements())
  {
    KnowLangTierToken entty = exprEntts.nextElement();
    if (entty.getName().equals("ID"))
      sExpression += " " + oKBTraversal.evaluateID(entty);
    else  //SINGLE_TOKEN, INT_VAL, etc.
      sExpression += " " + oKBTraversal.evaluateSingleToken(entty);
  }

  ScriptEngineManager mgr = new ScriptEngineManager();
  ScriptEngine engine = mgr.getEngineByName("JavaScript");

  if ( sExpression.trim().isEmpty() ) sExpression = "false";
  evaluatedStateExpression = sExpression;
  try
  {
    stateEvaluation = (Boolean) engine.eval(sExpression);
  } catch (ScriptException e)
  {
    stateEvaluation = false;
  }
  setStateEvaluation(stateEvaluation);
  return stateEvaluation;
}
```

As shown in the code above, in its first part, the method *evaluateState()* creates the string *sExpression* of the Boolean expression by evaluating concepts' values. This is done by the KBTraversal's method *oKBTraversal.evaluateID()*. In its second part, the method evaluates the already built string *sExpression* by using the Java Script engine. As mentioned above, in case of an error, the *evaluationState()* method returns *false*, i.e., the state is evaluated as *non-active*.

The third example presents a method called *reportBehaviorResult()* that is intended to handle the $TELL\_BEHAVIOR$ operator. Recall that when the reasoner reasons about a self-adaptive behavior, it determines the most appropriate policy first and then the most appropriate sequence of actions via that policy's mappings (see Section 2.2). Every behavior determined by the reasoner has a unique *signature*, which can be used by the host system to report to the reasoner whether a behavior has been successful. In that context, the *reportBehaviorResult()* method adds on the learning and self-adaptive abilities of the KnowLang Reasoner. The method is implemented by the *ExtendedKLPolicy* class (see Figure 10) and basically, it recomputes the probabilities embeded in the policy's mappings, which leads to *reinforcement learning* [VHM+12].

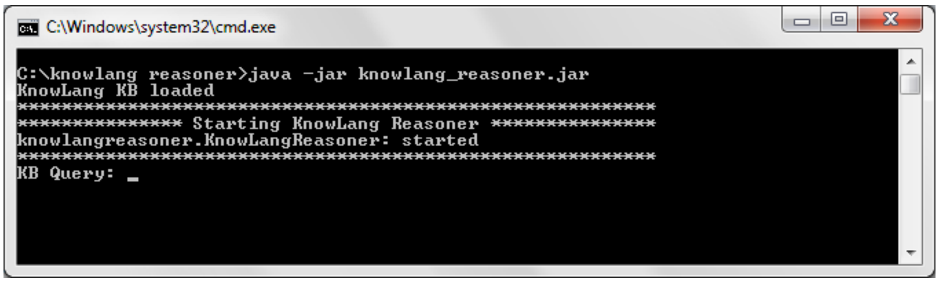
```
public void reportBehaviorResult(String pBehaviorSignature, boolean pSuccesful)
{
  double step = KnowLangReasonerDef.probabilityStep;
  if ( !pSuccesful ) step = -step;

  KLPolicyMapping policyMapping = findPolicyMapping(pBehaviorSignature);
  if ( policyMapping != null )
  {
    double newProbability = policyMapping.getProbabilityDouble() + step;
    policyMapping.setProbability("" + newProbability);
  }
}
```

As shown by the code above, a globally defined probability step (*KnowLangReasonerDef.probabilityStep*) is added to or deducted from the current probability rate of the mapping having the same behavior signature (*pBehaviorSignature*). In that way, by changing the probability rates of the policy's mappings, the reasoner basically changes the behavior preferences and self-adaptation emerges as shifting from one sequence of action to another one when behavior is successful or not. Recall that similar self-adaptation emerges at the level of situation-policy relations as well (see Section 2.2). This mechanism though, is implemented by the *KLLearner* class, which implements a method that computes this property in the reasoner's learning phase and uses it later to decide what policy to run in order to respond to a particular situation. Moreover, in the learning phase, the reasoner may also decide to re-compute the *KnowLangReasonerDef.probabilityStep* property, which will either increase or decrease the speed of learning.

5.4 Awareness with KnowLang

In this exercise, we performed tests with the KnowLang Reasoner to simulate awareness emerging when the reasoner operates over the eMobility KB. To perform the exercise, we implemented a special host application running the KnowLang Reasoner and communicating with it via a command line where users enter ASK and TELL commands. In that way, the host application communicates with the KnowLang interpreter implemented by the *KBQueryInterpreter* class (see Section 5.3.1) to convert the human-friendldy commands to their ASK and TELL counterparts.

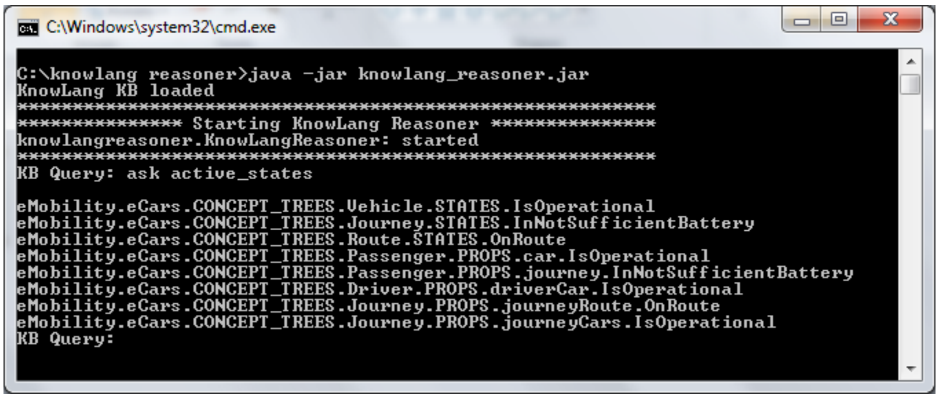
In the first phase of the awareness simulation, we run the host application, which in turn loaded and run the KnowLang Reasoner. Next, the KnowLang Reasoner performed the startup process as described in Section 5.3.1. Figure 11 depicts the result screen of Phase 1.



```
C:\knowlang reasoner>java -jar knowlang_reasoner.jar
KnowLang KB loaded
************************************************************
*************** Starting KnowLang Reasoner ***************
knowlangreasoner.KnowLangReasoner: started
************************************************************
KB Query: _
```

Figure 11: Awareness Simulation Test: Phase 1

Note that at the end of that phase, the eMobility KB is loaded and the reasoner waits for further instructions. When in a “waiting” mode, the reasoner is iterating over the awareness control loop as described in 5.3.1. Therefore, all the states expressed in KnowLang are constantly evaluated at any loop iteration (performed by the analyzer), which leads to re-evaluation of all the goals and situations expressed with states (see Section 2.2).

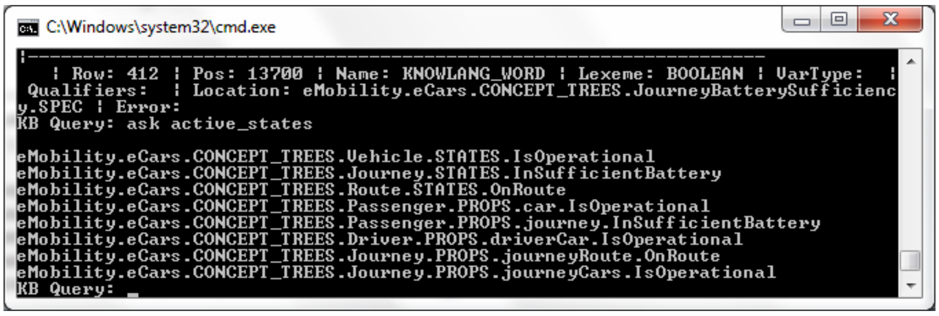


```
C:\knowlang reasoner>java -jar knowlang_reasoner.jar
KnowLang KB loaded
************************************************************
*************** Starting KnowLang Reasoner ***************
knowlangreasoner.KnowLangReasoner: started
************************************************************
KB Query: ask active_states

eMobility.eCars.CONCEPT_TREES.Vehicle.STATES.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.STATES.InNotSufficientBattery
eMobility.eCars.CONCEPT_TREES.Route.STATES.OnRoute
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.car.IsOperational
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.journey.InNotSufficientBattery
eMobility.eCars.CONCEPT_TREES.Driver.PROPS.driverCar.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.OnRoute
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyCars.IsOperational
KB Query:
```

Figure 12: Awareness Simulation Test: “ask active_states” Command

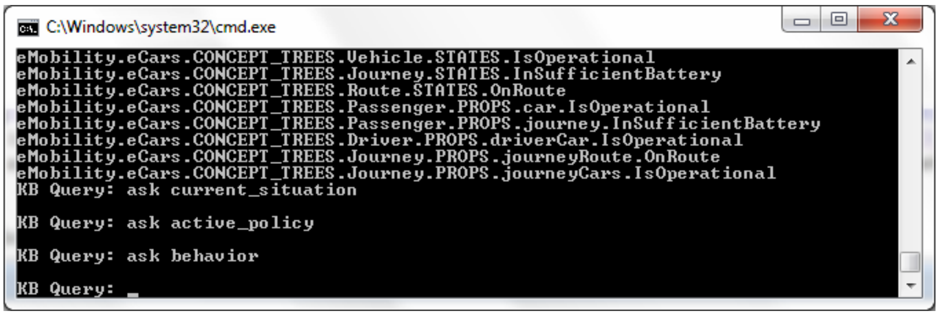
5.4.1 Insufficient Battery Awareness Simulation

In this test we simulated battery insufficiency to accomplish a planned journey. Here, to see the current state of the system, we asked the reasoner about all active states (see Figure 12). As shown in Figure 12, the reasoner responded by returning a few currently active states, including the *eMobility.eCars.CONCEPT_TREES.Journey.STATES.InNotSufficientBattery*. A closer look at the KnowLang specification of this state will show that the state's expression involves the evaluation of its neighbor state, which in turn evaluates the *JourneyBatterySufficiency* metric (see Section 3.2). Recall that metrics represent sensors in KnowLang [VHM+12]. Here, to test the awareness of new sensory data, we updated that metric's value and asked for the active states again (see Figure 13).



```
|----------------------------------------------------------------------------
   | Row: 412 | Pos: 13700 | Name: KNOWLANG_WORD | Lexeme: BOOLEAN | VarType:  |
 Qualifiers:  | Location: eMobility.eCars.CONCEPT_TREES.JourneyBatterySufficienc
y.SPEC | Error:
KB Query: ask active_states

eMobility.eCars.CONCEPT_TREES.Vehicle.STATES.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.STATES.InSufficientBattery
eMobility.eCars.CONCEPT_TREES.Route.STATES.OnRoute
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.car.IsOperational
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.journey.InSufficientBattery
eMobility.eCars.CONCEPT_TREES.Driver.PROPS.driverCar.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.OnRoute
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyCars.IsOperational
KB Query: _
```

Figure 13: Awareness Simulation Test: "tell sensor_val JourneyBatterySufficiency true" and "ask active_states" Commands

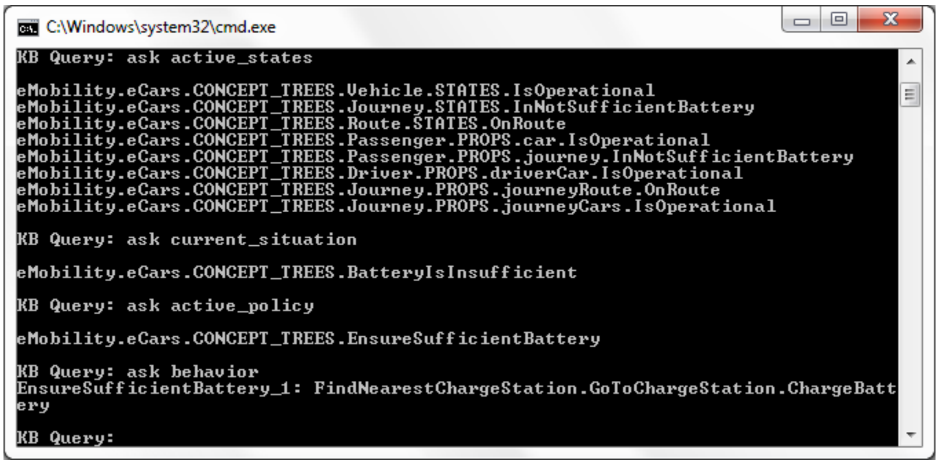


```
eMobility.eCars.CONCEPT_TREES.Vehicle.STATES.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.STATES.InSufficientBattery
eMobility.eCars.CONCEPT_TREES.Route.STATES.OnRoute
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.car.IsOperational
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.journey.InSufficientBattery
eMobility.eCars.CONCEPT_TREES.Driver.PROPS.driverCar.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.OnRoute
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyCars.IsOperational
KB Query: ask current_situation

KB Query: ask active_policy

KB Query: ask behavior

KB Query: _
```

Figure 14: Awareness Simulation Test: "ask current_situation", "ask active_policy", and "ask behavior" Commands

As shown in Figure 13, the *InNotSufficientBattery* state was not active anymore, but the *InSufficientBattery* state appeared as active, which is correct considering the specification of both states (see Section 3.2). Next, we asked the reasoner to see if there is a *current situation* that requires self-adaptation, *active policy*, and *recommended behavior*. As shown in Figure 14, the reasoner did not determine any of those.

In the next step, we simulated another sensory input (*tell sensor_val JourneyBatterySufficiency false*) that activated the *InNotSufficientBattery* state as expected. Then, we asked the reasoner again about the presence of a situation that requires self-adaptation, *active policy*, and *recommended behavior*. As shown in Figure 15, this time the reasoner responded by determining the current situation as *BatteryIsInsufficient*, which is actually specified over the *InNotSufficientBattery* state, i.e., that situation is present when the state is active. Moreover, the reasoner determined the *EnsureSufficientBattery* policy as active, and consecutively, it returned the recommended behavior as *EnsureSufficientBattery_1: FindNearestChargeStation.GoToChargeStation.ChargeBattery*. Here, the active policy is as-



```
KB Query: ask active_states

eMobility.eCars.CONCEPT_TREES.Vehicle.STATES.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.STATES.InNotSufficientBattery
eMobility.eCars.CONCEPT_TREES.Route.STATES.OnRoute
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.car.IsOperational
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.journey.InNotSufficientBattery
eMobility.eCars.CONCEPT_TREES.Driver.PROPS.driverCar.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.OnRoute
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyCars.IsOperational

KB Query: ask current_situation

eMobility.eCars.CONCEPT_TREES.BatteryIsInsufficient

KB Query: ask active_policy

eMobility.eCars.CONCEPT_TREES.EnsureSufficientBattery

KB Query: ask behavior
EnsureSufficientBattery_1: FindNearestChargeStation.GoToChargeStation.ChargeBatt
ery

KB Query:
```

Figure 15: Awareness Simulation Test: "tell sensor_val JourneyBatterySufficiency false", "ask active_states", "ask current_situation", "ask active_policy", and "ask behavior" Commands

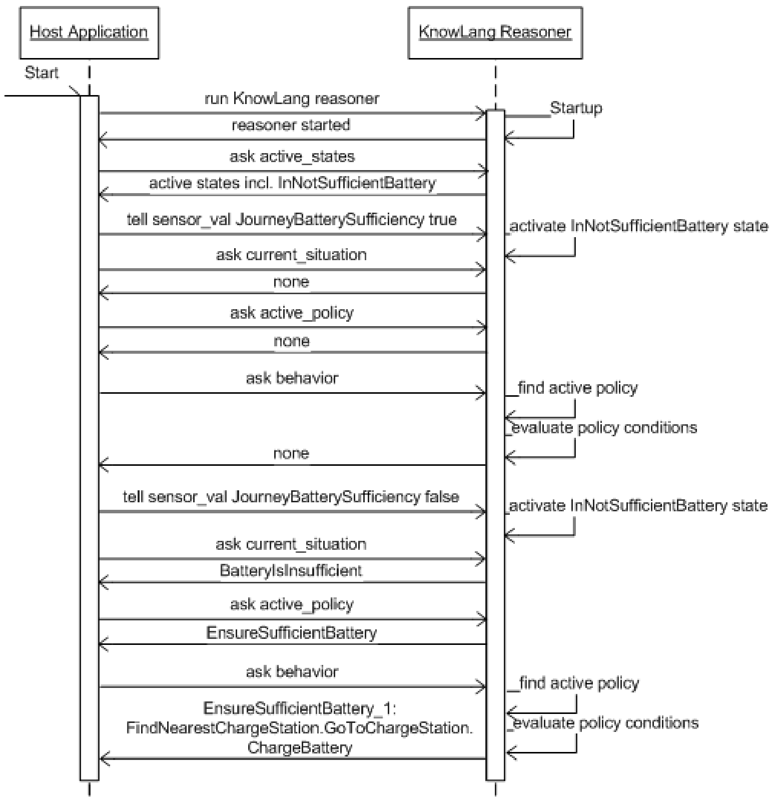

Host Application
KnowLang Reasoner
Start
run KnowLang reasoner
Startup
reasoner started
ask active_states
active states incl. InNotSufficientBattery
tell sensor_val JourneyBatterySufficiency true
activate InNotSufficientBattery state
ask current_situation
none
ask active_policy
none
ask behavior
find active policy
evaluate policy conditions
none
tell sensor_val JourneyBatterySufficiency false
activate InNotSufficientBattery state
ask current_situation
BatteryIsInsufficient
ask active_policy
EnsureSufficientBattery
ask behavior
find active policy
evaluate policy conditions
EnsureSufficientBattery_1: FindNearestChargeStation.GoToChargeStation. ChargeBattery

Figure 16: Awareness Process Steps

sociated with the current behavior, and the reasoner recommends a sequence of actions:

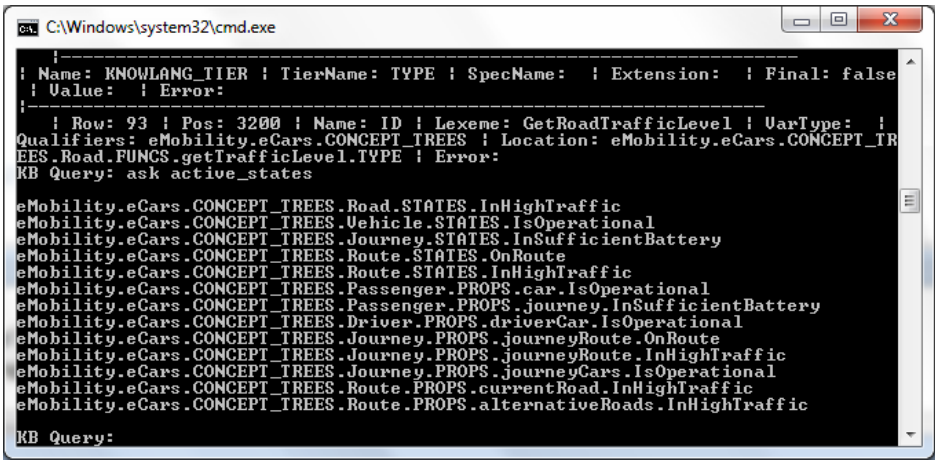
FindNearestChargeStation⇒GoToChargeStation⇒ChargeBattery

that will eventually lead the system out of the present situation. Note that the first string in the returned behavior is the behavior signature *EnsureSufficientBattery_1*. Recall that the reasoner grands each determined behavior with a unique signature that can be used by the *TELL_BEHAVIOR* operator to report how successful the recommended behavior is (see Section 5.3.2).

Figure 16 depicts a UML sequence diagram presenting the entire simulation process in steps as described above.

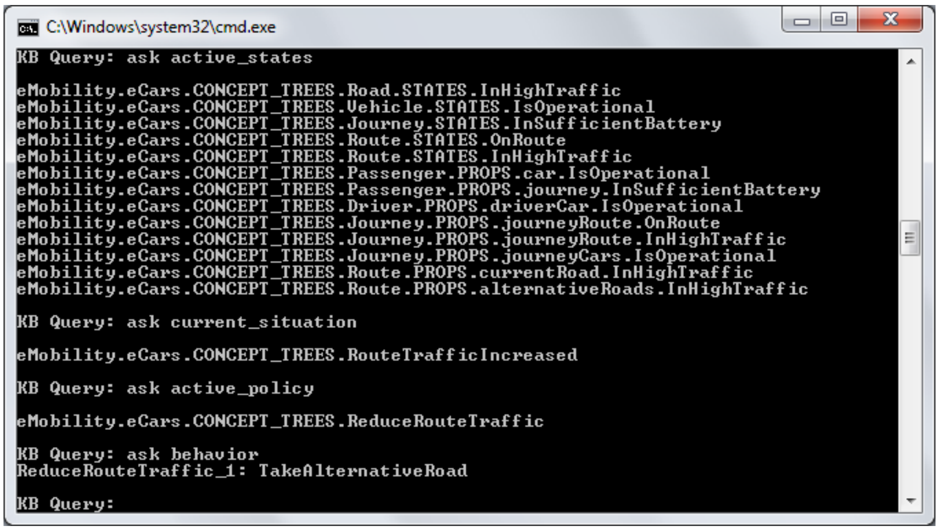
5.4.2 High Traffic Awareness and Reinforcement Learning Simulation

In the next test of our awareness simulation, we simulated a high traffic situation, which requires self-adaptation. To do so, we activated a state (*eMobility.eCars.CONCEPT_TREES.Route.STATES.InHighTraffic*) that in turn, made the *RouteTrafficIncreased* situation present (for the KnowLang specification of that situation see Section 3.3).



```
   |-----------------------------------------------------------------
| Name: KNOWLANG_TIER | TierName: TYPE | SpecName:   | Extension:   | Final: false
 | Value:   | Error:
|-----------------------------------------------------------------
   | Row: 93 | Pos: 3200 | Name: ID | Lexeme: GetRoadTrafficLevel | VarType:  |
Qualifiers: eMobility.eCars.CONCEPT_TREES | Location: eMobility.eCars.CONCEPT_TR
EES.Road.FUNCS.getTrafficLevel.TYPE | Error:
KB Query: ask active_states

eMobility.eCars.CONCEPT_TREES.Road.STATES.InHighTraffic
eMobility.eCars.CONCEPT_TREES.Vehicle.STATES.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.STATES.InSufficientBattery
eMobility.eCars.CONCEPT_TREES.Route.STATES.OnRoute
eMobility.eCars.CONCEPT_TREES.Route.STATES.InHighTraffic
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.car.IsOperational
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.journey.InSufficientBattery
eMobility.eCars.CONCEPT_TREES.Driver.PROPS.driverCar.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.OnRoute
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.InHighTraffic
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyCars.IsOperational
eMobility.eCars.CONCEPT_TREES.Route.PROPS.currentRoad.InHighTraffic
eMobility.eCars.CONCEPT_TREES.Route.PROPS.alternativeRoads.InHighTraffic

KB Query:
```

Figure 17: Awareness Simulation Test: "tell func_val road getTrafficLevel 71" and "ask active_states" Commands



```
KB Query: ask active_states

eMobility.eCars.CONCEPT_TREES.Road.STATES.InHighTraffic
eMobility.eCars.CONCEPT_TREES.Vehicle.STATES.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.STATES.InSufficientBattery
eMobility.eCars.CONCEPT_TREES.Route.STATES.OnRoute
eMobility.eCars.CONCEPT_TREES.Route.STATES.InHighTraffic
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.car.IsOperational
eMobility.eCars.CONCEPT_TREES.Passenger.PROPS.journey.InSufficientBattery
eMobility.eCars.CONCEPT_TREES.Driver.PROPS.driverCar.IsOperational
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.OnRoute
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyRoute.InHighTraffic
eMobility.eCars.CONCEPT_TREES.Journey.PROPS.journeyCars.IsOperational
eMobility.eCars.CONCEPT_TREES.Route.PROPS.currentRoad.InHighTraffic
eMobility.eCars.CONCEPT_TREES.Route.PROPS.alternativeRoads.InHighTraffic

KB Query: ask current_situation

eMobility.eCars.CONCEPT_TREES.RouteTrafficIncreased

KB Query: ask active_policy

eMobility.eCars.CONCEPT_TREES.ReduceRouteTraffic

KB Query: ask behavior
ReduceRouteTraffic_1: TakeAlternativeRoad

KB Query:
```

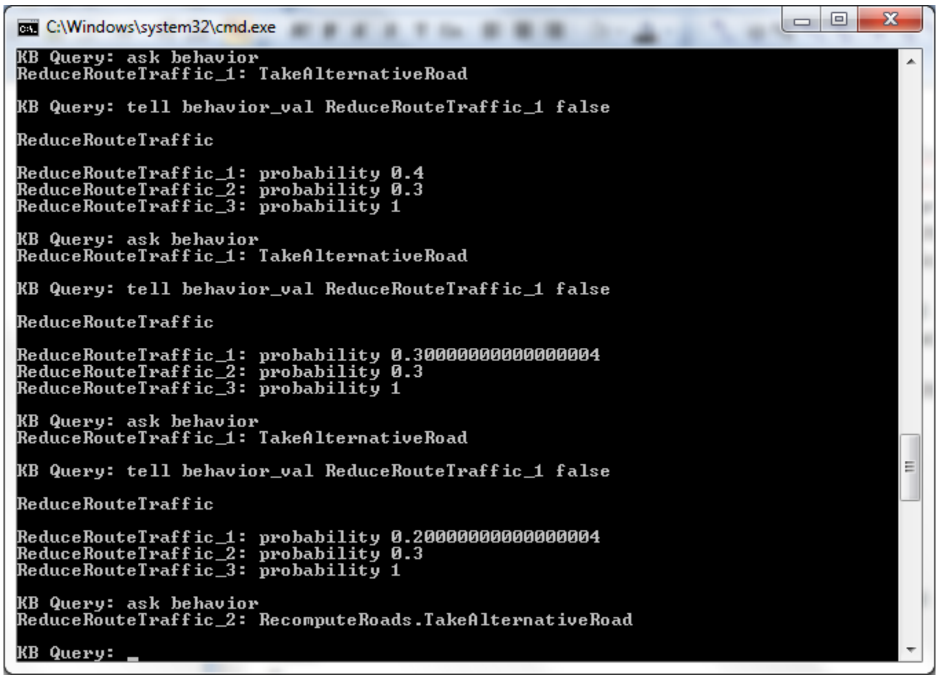
Figure 18: Awareness Simulation Test: "ask active_states", "ask current_situation", "ask active_policy", and "ask behavior" Commands

To activate that state, we supplied the reasoner with the necessary data as following:

tell func_val road getTrafficLevel 71

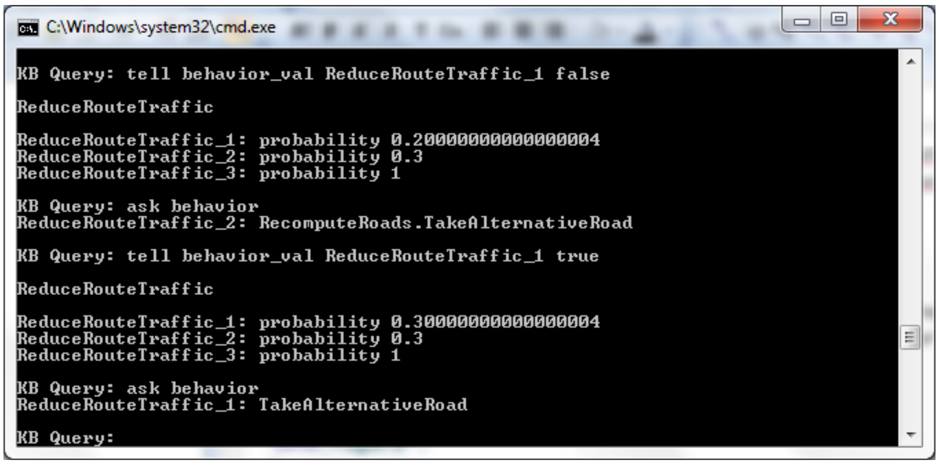
With the *tell* operator above, we provided the function *getTrafficLevel* of the *road* concept with a value that will make the *InHighTraffic* state active (see Figure 17).

In the next step, we asked the reasoner about the presence of situation that requires self-adaptation, *active policy*, and *recommended behavior*. As shown in Figure 18, the reasoner responded by determining the current situation as *RouteTrafficIncreased*, which is actually specified over the *InHighTraffic* state, i.e., that situation is present when the state is active.



```
C:\Windows\system32\cmd.exe
KB Query: ask behavior
ReduceRouteTraffic_1: TakeAlternativeRoad

KB Query: tell behavior_val ReduceRouteTraffic_1 false

ReduceRouteTraffic

ReduceRouteTraffic_1: probability 0.4
ReduceRouteTraffic_2: probability 0.3
ReduceRouteTraffic_3: probability 1

KB Query: ask behavior
ReduceRouteTraffic_1: TakeAlternativeRoad

KB Query: tell behavior_val ReduceRouteTraffic_1 false

ReduceRouteTraffic

ReduceRouteTraffic_1: probability 0.30000000000000004
ReduceRouteTraffic_2: probability 0.3
ReduceRouteTraffic_3: probability 1

KB Query: ask behavior
ReduceRouteTraffic_1: TakeAlternativeRoad

KB Query: tell behavior_val ReduceRouteTraffic_1 false

ReduceRouteTraffic

ReduceRouteTraffic_1: probability 0.20000000000000004
ReduceRouteTraffic_2: probability 0.3
ReduceRouteTraffic_3: probability 1

KB Query: ask behavior
ReduceRouteTraffic_2: RecomputeRoads.TakeAlternativeRoad

KB Query: _
```

Figure 19: Awareness Simulation Test: "tell behavior_val ReduceRouteTraffic_1 false", "ask behavior" Commands



```
C:\Windows\system32\cmd.exe
KB Query: tell behavior_val ReduceRouteTraffic_1 false

ReduceRouteTraffic

ReduceRouteTraffic_1: probability 0.20000000000000004
ReduceRouteTraffic_2: probability 0.3
ReduceRouteTraffic_3: probability 1

KB Query: ask behavior
ReduceRouteTraffic_2: RecomputeRoads.TakeAlternativeRoad

KB Query: tell behavior_val ReduceRouteTraffic_1 true

ReduceRouteTraffic

ReduceRouteTraffic_1: probability 0.30000000000000004
ReduceRouteTraffic_2: probability 0.3
ReduceRouteTraffic_3: probability 1

KB Query: ask behavior
ReduceRouteTraffic_1: TakeAlternativeRoad

KB Query:
```

Figure 20: Awareness Simulation Test: "tell behavior_val ReduceRouteTraffic_1 true", "ask behavior" Commands

The reasoner also determined the *ReduceRouteTraffic* policy as active, and consecutively, it returned the recommended behavior as *ReduceRouteTraffic_1: TakeAlternativeRoad*. Here, the active policy is associated with the current behavior, and the reasoner recommends the action *TakeAlternativeRoad* that will eventually lead the system out of the present situation, i.e., it will reduce the traffic. Please consult Section 3.3 for details on the specification of both the situation and policy.

Next, to simulate reinforcement learning, we simulated a few consecutive fails of the *ReduceRouteTraffic_1* behavior by feeding the reasoner with simulated *negative* feedback of that behavior's execution as following:

tell behavior_val ReduceRouteTraffic_1 false

As shown in Figure 19, the reasoner switched to another recommended behavior (*ReduceRouteTraffic_2: RecomputeRoads.TakeAlternativeRoad*) after a few fails of the *ReduceRouteTraffic_1* behavior. This behavior switch was possible because of the reinforcement learning ability of the KnowLang Reasoner (see Section 2.2). Next to demonstrate that the learning ability works in both directions, we simulated increase of the confidence in the *ReduceRouteTraffic_1* behavior by feeding the reasoner with simulated *positive* feedback of that behavior's execution as shown in Figure 20. In that case, the reasoner switched back to its initial recommended behavior (*ReduceRouteTraffic_1*).

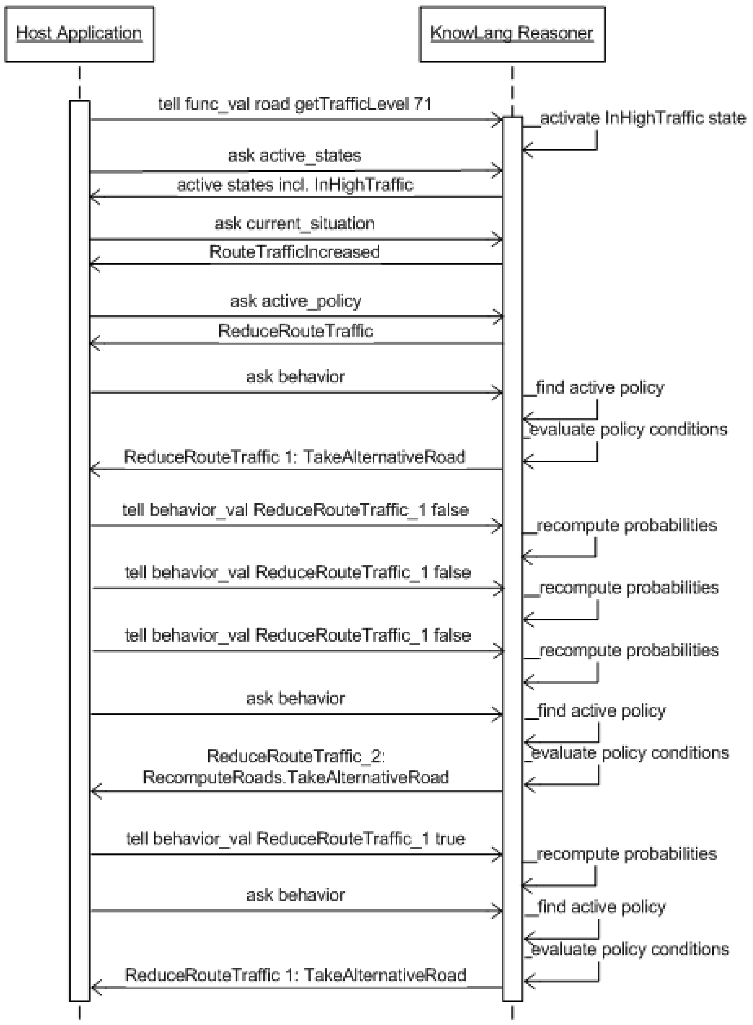

Host Application
KnowLang Reasoner
tell func_val road getTrafficLevel 71
activate InHighTraffic state
ask active_states
active states incl. InHighTraffic
ask current_situation
RouteTrafficIncreased
ask active_policy
ReduceRouteTraffic
ask behavior
find active policy
evaluate policy conditions
ReduceRouteTraffic 1: TakeAlternativeRoad
tell behavior_val ReduceRouteTraffic_1 false
recompute probabilities
tell behavior_val ReduceRouteTraffic_1 false
recompute probabilities
tell behavior_val ReduceRouteTraffic_1 false
recompute probabilities
ask behavior
find active policy
evaluate policy conditions
ReduceRouteTraffic_2: RecomputeRoads.TakeAlternativeRoad
tell behavior_val ReduceRouteTraffic_1 true
recompute probabilities
ask behavior
find active policy
evaluate policy conditions
ReduceRouteTraffic 1: TakeAlternativeRoad

Figure 21: Awareness and Reinforcement Learning Process Steps

Note that in the tests above, although, as shown in figures 19 and 20, the *ReduceRouteTraffic_3* has the highest probability rate, it was never recommended by the reasoner, because its starting conditions were not met (see Section 3.3).

Figure 21 depicts a UML sequence diagram presenting the entire simulation process in steps as described above.

6 Summary and Future Goals

In the course of the fourth year of WP3, we completed the implementation of the KnowLang Framework and successfully built the knowledge models for all the three ASCENS case studies. Moreover, we completed the first version of the KnowLang Reasoner. Although subject of further enhancement, the current implementation of the KnowLang Reasoner is complete and comprehensive enough to provide a powerful mechanism for self-adaptive reasoning. The reasoner runs in the context of self-adaptive systems and operates over a KnowLang-specified, yet KnowLang-compiled KB. It provides a predefined set of ASK and TELL operators allowing for communication with that KB. TELL operators feed the reasoner with important information so it can update the KB with recent changes in both the system and execution environment. The system running the reasoner 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. Finally, ASK operators may provide the system with awareness-based conclusions about current situations, states, etc.

As a proof-of-concept test case, we used the KnowLang Reasoner to simulate awareness in the ASCENS case studies. In this deliverable, we have presented how the KnowLang Reasoner, operating over the eMobility KB, responded to queries about active states, current situations, and active policies. Moreover, the test results demonstrated how the KnowLang Reasoner reasons about situations and recommends behavior (a sequence of actions) that eventually may drive the system out of a particular situation. Finally, we have demonstrated that the reasoner is capable of reinforcement learning based on the past experience of recommended behavior.

Our plans for future work are mainly concerned with further development of both the KnowLang Framework and KnowLang Reasoner, along with their further integration in the ARE Framework Toolset. Concerning the KnowLang Framework, we plan to develop constructs for knowledge representation of special atomic and sharable *state signs* that should help the reasoner reason on upcoming activation of various states. This will help the reasoner predict in a more accurate fashion the necessary self-adaptive behavior. Other upcoming development activities are concerned with the implementation of the KnowLang GENERATE_NEXT_ACTIONS operator, which is based on the computations performed by a special reward function that needs to be implemented by the KnowLang Reasoner. The KnowLang Reward Function (KLRF) observes the outcome of the actions to compute the possible successor states of every possible action execution and grants the actions with special reward number considering the current system state (or states, if the current state is a composite state) and goals. KLRF is based on past experience and uses Discrete Time Markov Chains for probability assessment after action executions.

To conclude, we would like to say that we believe WP3 achieved great results, which solely and uniquely demonstrate how successful this WP is.

References

[BBR+11] M. Bonani, T. Baaboura, P. Retornaz, F. Vaussard, S. Magnenat, D. Burnier, V. Longchamp, and F. Mondada. marXbot, Laborotoire de Systemes Robotiques (LSRO), Ecole Polytechnique Federale de Lausanne. mobots.epfl.ch, 2011. http://mobots.epfl.ch/marxbot.html.

[BFZ14] Nicola Bicocchi, Damiano Fontana, and Franco Zambonelli. A self-aware, reconfigurable architecture for context awareness. In *IEEE Symposium on Computers and Communications*, Madeira, Portugal, 2014.

[BVZH14] N. Bicocchi, E. Vassev, F. Zambonelli, and M. Hinchey. Reasoning on data streams: an approach to adaptation in pervasive systems. In P. C. Vinh, E. Vassev, and M. Hinchey, editors, *Nature of Computation and Communication, Lecture Notes of the Institute for Computer Sciences, Volume 144*. Springer, 2014.

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

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

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

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

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

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

[Vas12] E. Vassev. Operational semantics for KnowLang ASK and TELL operators. Technical Report Lero-TR-2012-05, Lero, University of Limerick, Ireland, 2012.

[VH13a] E. Vassev and M. Hinchey. Autonomy requirements engineering. *IEEE Computer*, 46(8):82–84, 2013.

[VH13b] E. Vassev and M. Hinchey. Autonomy requirements engineering. In *Proceedings of the 14th IEEE International Conference on Information Reuse and Integration (IRI 2013)*, pages 175–184. IEEE Computer Society, 2013.

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

[VH13d] E. Vassev and M. Hinchey. On the autonomy requirements for space missions. In *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing Workshops (ISCORCW 2013)*. IEEE Computer Society, 2013.

[VH14] E. Vassev and M. Hinchey. Modeling swarm robotics with KnowLang. In P. C. Vinh, E. Vassev, and M. Hinchey, editors, *Nature of Computation and Communication, Lecture Notes of the Institute for Computer Sciences, Volume 144*. Springer, 2014.

[VH15] E. Vassev and M. Hinchey. Knowlang: Knowledge representation for self-adaptive systems. *IEEE Computer*, 48(2), 2015.

[VHBH14] E. Vassev, N. Hoch, H. Bensler, and M. Hinchey. Formalizing eMobility with KnowLang. In *Proceedings of C* Conference on Computer Science & Software Engineering (C3S2E'14)*, pages 27–34. ACM, 2014.

[VHBM13] E. Vassev, M. Hinchey, N. Bicocchi, and P. Mayer. D3.3: Third Report on WP3: Knowledge Modeling for ASCENS Case Studies and KnowLang Implementation, 2013. ASCENS Deliverable.

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