KnowLang – A Formal Specification Model for Self-adaptive Systems

Mike Hinchey and Emil Vassev(✉)

Lero–The Science Foundation Ireland Research Centre for Software, University of Limerick, Limerick, Ireland
mike.hinchey@lero.ie, emil.i.vassev@ul.ie

Abstract. KnowLang is a framework for knowledge representation and reasoning (KR&R) that aims at efficient and comprehensive knowledge structuring and awareness based on logical and statistical reasoning. It tackles both explicit representation of domain concepts and relationships and explicit representation of particular and general factual knowledge, in terms of predicates, names, connectives, quantifiers and identity. Moreover, it handles uncertain knowledge in which additive probabilities are used to represent degrees of belief. Other remarkable features are related to knowledge cleaning and knowledge representation for autonomic self-adaptive behaviour. Knowledge specified with KnowLang takes the form of a Knowledge Base (KB) that outlines a KR context. A special KnowLang Reasoner operates in this context to allow for knowledge querying and update. In addition, the reasoner can infer special self-adaptive behaviour.

At its very core, KnowLang is a formal specification language providing a comprehensive specification model aiming at addressing the knowledge representation problem of self-adaptive systems. The complexity of the problem necessitated the use of a specification model where knowledge can be presented at different levels of abstraction and grouped by following both hierarchical and functional patterns. In this paper, we outline the formal semantics of the KnowLang multi-tier specification model. The model is outlined in terms of layers dedicated to knowledge corpuses, KB operators, and inference primitives.

Keywords: KnowLang · self-adaptive systems · formal specification

1 Introduction

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

KnowLang [14,15,17–20] is a framework for KR&R that aims at efficient and comprehensive knowledge structuring and awareness based on *logical* and *statistical reasoning*. It helps us to tackle 1) explicit representation of domain concepts and relationships; 2) explicit representation of particular and general factual knowledge, in terms of predicates, names, connectives, quantifiers and identity; and 3) uncertain knowledge in which additive probabilities are used to represent degrees of belief. Other remarkable features are related to knowledge cleaning (allowing for efficient reasoning) and knowledge representation for autonomic self-adaptive behaviour. Knowledge specified with KnowLang takes the form of a Knowledge Base (KB) that outlines a KR context. A special KnowLang Reasoner operates in this context to allow for knowledge querying and update. In addition, the reasoner can infer special self-adaptive behaviour.

The rest of this paper is organized as follows. Section 2 presents the KnowLang formal specification model including the constructs for specifying self-adaptive behaviour. Section 3 provides a discussion on how KnowLang copes with challenging problems such as encoded versus represented knowledge, the specification of states, situations, goals and policies, and how sensory data is converted to KR symbols. Section 4 outlines the KnowLang syntax. Section 5 provides and example of KR for Self-adaptive Behaviour with KnowLang and Sect. 6 describes a case study where KnowLang has been used to specify and formalize an eMobility autonomous system. Finally, Sect. 7 provides brief concluding remarks and a summary of our future goals.

2 Specification Model

At its very core, KnowLang is a formal specification language providing a comprehensive specification model aiming at addressing the knowledge representation problem for self-adaptive systems. The complexity of the problem necessitated the use of a *specification model* (inspired by the ASSL's specification model [11]) where knowledge can be presented at different levels of abstraction and grouped by following both hierarchical and functional patterns. KnowLang imposes a multi-tier specification model (see Fig. 1), where we specify a KB composed of layers dedicated to *knowledge corpuses*, *KB (knowledge base) operators* and *inference primitives*.

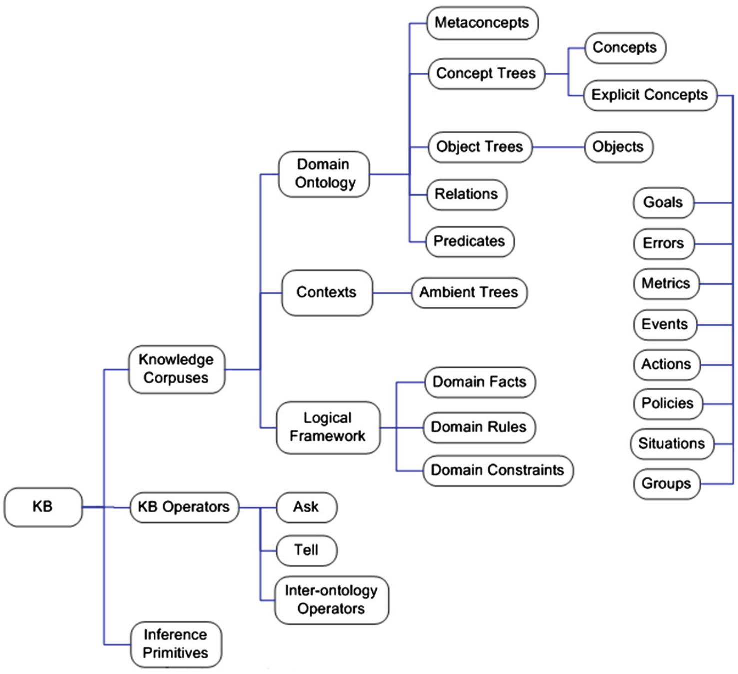

Fig. 1. KnowLang Specification Model

Definitions 1 through 58 outline a BNF-like [6] formal representation of the KnowLang Specification Model. As shown in Definition 1, a Knowledge Base is a tuple of three main knowledge components – *knowledge corpus* (Kc), *KB operators* (Op) and *inference primitives* (Ip). A Kc is a tuple of three knowledge components – *ontologies* (O), *contexts* (Cx) and *logical framework* (Lf) (see Definition 2). Further, a domain ontology is composed of hierarchically organized sets of *meta-concepts* (Cm), *concept trees* (Ct), *object trees* (Ot), *relations* (R) and *predicates* (V) (see Definition 4). Note that the trees in our model (e.g., concept trees, object trees, etc.) can be direct acyclic graphs. Moreover, note that in the definitions below we denote a finite set of elements El with $\{el_1, el_2, \ldots, el_n\}, n \geq 0$ where by omitting el_0 we allow an empty set, e.g., see the definition of meta-concepts (Cm) 5.

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

FORMAL REPRESENTATION OF KNOWLANG

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

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

DOMAIN ONTOLOGIES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The KnowLang policies (Π) drive the autonomic behaviour of the system. A policy π has a *goal* (g), *policy situations* (Si_π), *policy-situation relations* (R_π), and *policy conditions* (N_π) mapped to *policy actions* (A_π) where the evaluation of N_π may eventually (with some degree of probability) imply the evaluation of actions (denoted with $N_\pi \overset{[Z]}{\to} A_\pi$) (see Definition 19).

A condition is a Boolean expression over ontology (see Definition 21), e.g., the occurrence of a certain event. *Policy situations* Si_π are situations (see Definition 25) that may trigger (or imply) a policy π, in compliance with the policy-situations relations R_π(denoted with $Si_\pi \overset{[R_\pi]}{\to} \pi$), thus implying the evaluation of the policy conditions N_π(denoted with $\pi \to N_\pi$)(see Definition 19). A policy may comprise optional policy-situation relations (R_π) justifying the relationships between a policy and the associated situations. The presence of probabilistic beliefs in both *mappings* and *policy relations* justifies the probability of policy execution, which may vary with time. Note that Sect. 5 discusses in detail how the KR of policies, situations and relations provides for self-adaptive behaviour.

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

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

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

Definition 19. $\pi := <g, Si_\pi, [R_\pi], N_\pi, A_\pi, map(N_\pi, A_\pi, [Z])>$ *(Policy)*

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

$Si_\pi \subset Si, Si_\pi := \{si_{\pi_1}, si_{\pi_2}, \ldots, si_{\pi_n}\}, n \geq 0$ *(Si_π – Policy Situations)*

$R_\pi \subset R, R_\pi := \{r_{\pi_1}, r_{\pi_2}, \ldots, r_{\pi_n}\}, n \geq 0$ *(R_π-Policy-Situation Relations)*

$\forall r_\pi \in R_\pi \bullet (r_\pi := <si_\pi, [rn], [Z], \pi>), si_\pi \in Si_\pi$

$Si_\pi \overset{[R_\pi]}{\to} \pi \to N_\pi$ *(Policy situations may imply the policy they are related to)*

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

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

Definition 22. $g := \langle \Rightarrow s' \rangle | \langle s \Rightarrow s' \rangle$ *(Goal)*

$\Rightarrow s := \langle TELL \rhd ob.P \rangle | \langle TELL \rhd \{ob_0.P, ob_1.P, \ldots, ob_n.P\} \rangle | \langle TELL \rhd Er_s \rangle |$

$\langle TELL \rhd E_s \rangle | \langle TELL \rhd A_s \rangle$ *(State Transition)*

$Er_s \subset Er$, $E_s \subset E$, $A_s \subset A$ *(Er_s – State Errors, E_s – State Events, A_s – State Actions)*

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

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

Definition 25. $si := <s, A\overleftarrow{_{si}}, [E\overleftarrow{_{si}}], A_{si}>$ *(Situation)*

$A\overleftarrow{_{si}} \subset A$ *($A\overleftarrow{_{si}}$ – Executed Actions)*

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

$E\overleftarrow{_{si}} \subset E$ *($E\overleftarrow{_{si}}$ – Situation Events)*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Relations (R) connect two concepts (including predicates V), two objects, or an object with a concept and may have *probability distribution* Z (e.g., over time, over situations, over concepts' properties, etc.) (see Definition 38). A relation has an optional name, i.e., when the name is missing we have the implication relation. Probability distribution is provided to support *probabilistic reasoning*. By specifying relations with probability distributions we actually specify Bayesian Networks [7] connecting the concepts and objects of an ontology. Note that KnowLang considers binary relations only, but there could be multiple relations relating the same concepts/objects.

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

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

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

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

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

KNOWLANG CONTEXTS

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

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

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

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

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

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

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

The *ambient concepts* (see Definition 44) explicitly determine new level of deepness for their original concept tree, i.e., ambient concepts subsume all of their child concepts (if any). As result, when the system reasons about a particular context (expressed with ambient trees), the reasoning process does not consider those child concepts, but their ambient parents, which are far more generic, and thus less detailed. This technique reduces the size of the relevant knowledge, by temporarily removing from the concept trees all the ambient concepts' children (descendant concepts). We may think about ambient trees as filters the system applies at runtime to reduce the visibility of concepts of a concept tree. Note that this technique has been further developed in [16].

KNOWLANG LOGICAL FRAMEWORK

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

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

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

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

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

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

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

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

KNOWLEDGE BASE OPERATORS

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

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

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

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

INFERENCE PRIMITIVES

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

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

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

The Inference Primitives (Ip) (see Definition 58) are algorithms for reasoning and knowledge inference needed by the KnowLang Reasoner. These primitives are implementation (denoted with *impl* in Definition 58) of reasoning algorithms based on First Order Logic (FOL) [2] (and its extensions), First Order Probabilistic Logic (FOPL) [4] and Description Logics (DL) [1]. FOPL increases the power of FOL by allowing us to assert in a natural way "likely" features of objects and concepts via a probability distribution over the possibilities that we envision. Having logics with semantics gives us a notion of deductive entailment. Note that these algorithms together with the appropriate reasoning engines shall help the KnowLang Reasoner to query and update KB.

3 Meeting the Challenges

Both the KnowLang Specification Model and KnowLang Reasoner have been developed by taking into consideration some explicit challenges comprehensively described in our publications [17,20,21].

3.1 Encoded Versus Represented Knowledge

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

To properly represent the program implementation (classes, methods, etc.) in the KB, all the concepts and objects have an *IMPL Property* that relates a KnowLang structure to its program counterpart, if any. For example, a KnowLang concept might be specified with an IMPL property to link the concept to a program class or method. The following is the grammar definition supporting that [12].

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

3.2 States, Situations, Goals and Policies

A big challenge is *"how to express situations and reason about the same"*. Situations trigger self-adaptive behaviour (see Sect. 5) and it is very important to allow the reasoner to recognize them. To support this approach, KnowLang has introduced the *STATE explicit concepts* (see Definition 23 in Sect. 2). This helps each KnowLang concept to be specified with a set of important states the concept instances can be in. Thus, we explicitly specify a variety of states for important concepts (e.g., states "operational" and "non-operational" for the robot's Motion System). A KnowLang state is specified as a Boolean expression over ontology where we can use activation of events, execution of actions or changes in properties to build a state's Boolean expression [12]. Further, to facilitate the evaluation of complex states, we specify *PREDICATES* (see Definition 40 in Sect. 2). Complex states (e.g., system states) are the product of other states (e.g., the states of the system's components). States (usually system states) are also used to specify *GOALS*, another class of KnowLang explicit concepts (see Definition 22 in Sect. 2). Goals participate in the specification of KnowLang policies. A goal can be specified as a transition from a state to another. Recall that policies and situations participate in KnowLang relations (see Definition 19 in Sect. 2) that drive the *self-adaptive behaviour* (see Sect. 5). Therefore, because every situation is explicitly related to a state (a situation is determined by a state), it is relatively easy to check for the feasibility of a policy triggered by a specific situation, i.e., the policy's goal must have the same departing state as the situation's state.

3.3 Converting Sensory Data to KR Symbols

One of the biggest challenges is *"how to map sensory raw data to KR symbols"*. Our approach to this problem is to specify special explicit concepts called *METRICS* (see Definition 17 in Sect. 2). In general, a SCE system has sensors that connect it to the world and eventually help it to listen to its internal components.

These sensors generate raw data that represent the physical characteristics of the world. The problem is that these low-level data streams must be: 1) converted to programming variables or more complex data structures that represent collections of sensory data; 2) those programing data structures must be labeled with KR Symbols. Hence, it is required to relate encoded data structures with KR concepts and objects used for reasoning purposes. In our approach, we assume that each sensor is controlled by a software driver (e.g., specified in SCEL and implemented in Java) where appropriate methods are used to control the sensor and read data from it. Both the *sensory data* and *sensors* should be represented in the KB by using *METRIC* explicit concepts and instantiate objects of these concepts. By specifying a METRIC concept we introduce a *class of sensors* to the KB and by specifying objects, instances of that class, we give the actual KR of a real sensor. KnowLang allows the specification of four different types of metrics [12]:

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

4 KnowLang Syntax

We used the Backus-Naur Form (BNF) notation [6] to describe the syntax of the language and formally specify the KnowLang Grammar [12]. This helps the KnowLang framework to process sentences written in the KnowLang language. BNF [6] is a powerful meta-language that allows a context-free grammar specification. A partial presentation of the KnowLang Grammar in BNF is the following [12]:

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

As shown, the full KnowLang context-free grammar specification is obtained by the reduction of the (*KL-Spec -> bof Knowledge-Spec eof*) rule, which determines that a KB specified with KnowLang consists of *specification units*, each formed by a combination of *knowledge corpuses*, *KB operators* and *inference primitives*. Due to the complex structure of the KnowLang specification model (see Sect. 2) where each tier has its own structure, the complete KnowLang Grammar's specification cannot be presented here (please refer to [12] for the full KnowLang Grammar in BNF). Instead, we present an abstraction of the KnowLang Grammar, i.e., a meta-grammar. The following is a generic meta-grammar in Extended BNF [6] presenting the syntax rules for specifying KnowLang tiers.

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

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

5 KR for Self-adaptive Behaviour with KnowLang

KnowLang has intrinsic features supporting KR for autonomic systems. An *autonomic system* [5,13] is considered to be a self-adaptive system that changes its behaviour in response to stimuli from its execution and operational environment. Such behaviour is considered *autonomic* and *self-adaptive* [13] and is intended to drive a system in situations requiring adaptation. Any long-running system is subject to uncertainty in its execution environment due to potential changes in requirements, business conditions, available technology, etc. Thus, it is important to capture and cater for uncertainty as part of the development process. Failure to do so may result in systems that are too rigid to be fit for purpose, which is of particular concern for the domains that typically make use of self-adaptive technology. We hypothesize that modeling uncertainty and developing mechanisms for managing it as part of KR&R will lead to systems that are:

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

The ability to represent knowledge providing for *self-adaptive behaviour* is an important factor in dealing with uncertainty. In our approach, the autonomic self-adaptive behaviour is provided by *policies, events, actions, situations*, and *relations* between policies and situations (see Definitions 18 through 25 in Sect. 2).

Ideally, policies are specified to handle specific situations, which may trigger the application of policies. A policy exhibits a behaviour via actions generated in the environment or in the system itself. Specific conditions determine, which specific actions (among the actions associated with that policy – see Definition 19 in Sect. 2) shall be executed. These conditions are often generic and may differ from the situations triggering the policy. Thus, the behaviour not only depends on the specific situations a policy is specified to handle, but also depends on additional conditions. Such conditions might be organized in a way allowing for synchronization of different situations on the same policy. When a policy is applied, it checks what particular conditions are met and performs the associated actions via special mappings (see $map(N_\pi, A_\pi, [Z])$ in Definition 19 in Sect. 2). An optional probability distribution (Z) may additionally restrict the action execution. Although initially specified, the probability distribution at the mappings is recomputed after the execution of any involved action. The recomputation is based on the consequences of the action execution, which allows for reinforcement leaning.

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

As a proof of concept, we applied the approach to a case study on Ensemble of Robots. To illustrate autonomic behaviour based on this approach, let us suppose that we have a marXbot robot that carries items from point A to point B by using two possible routes – route one and route two (see Fig. 2).

A situation si_1: *"robot is in point A and loaded with items"* will trigger a policy π_1: *"go to point B via route one"* if the relation $r(si_1, \pi_1)$ has the higher probabilistic belief rate (let's assume that such a rate has been initially given to this relation because *route one* is shorter – see Fig. 2a). Any time when the robot gets into situation si_1 it will continue applying the π_1 policy until it gets into a situation si_2: *"route one is blocked"* while applying that policy. The si_2 situation will trigger a policy π_2: *"go back to si_1 and then apply policy π_3"* (see Fig. 2.b). Policy π_3 is defined as π_3: *"go to point B via route two"*. The

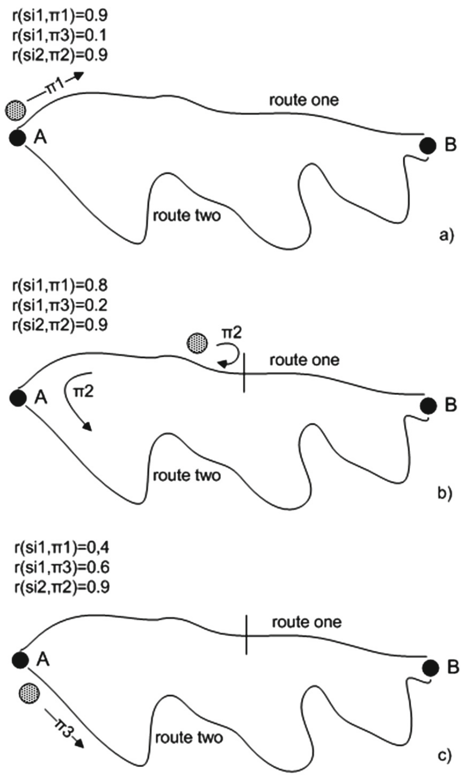

Fig. 2. A marXbot Self-adaptation Case Study

unsuccessful application of policy π_1 will decrease the probabilistic belief rate of relation $r(si_1, \pi_1)$ and the eventual successful application of policy π_3 will increase the probabilistic belief rate of relation $r(si_1, \pi_3)$ (see Fig. 2b). Thus, if route one continues to be blocked in the future, the relation $r(si_1, \pi_3)$ will get to have a higher probabilistic belief rate than the relation $r(si_1, \pi_1)$ and the robot will change its behaviour by choosing route two as a primary route (see Fig. 2c). Similarly, this situation can change in response to external stimuli, e.g., *route two* got blocked or a *"route one is obstacle-free"* message is received by the robot.

6 Formalizing eMobility with KnowLang

In eMobility, vehicles move according to a schedule defined by a driver [9,10]. 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.

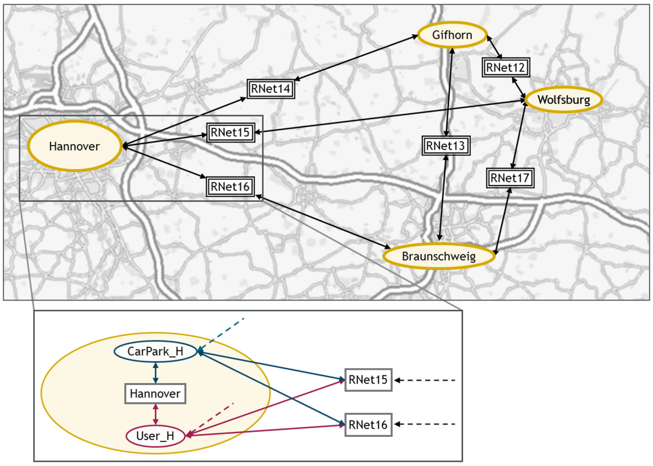

Fig. 3. eMobility Example [10]

Figure 3 shows a formal petri net representation of a real example scenario that considers four destinations (Wolfsburg, Gifhorn, Braunschweig, and Hannover), the road network between the destinations and the processes which are taking place at the destination locations [10]. 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).

In this constraint 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 [8]:

– *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 derived the eMobility goals along with the self-* objectives assisting these goals when self-adaptation is required. Note that the required analysis and process of building the goals model for eMobility along with the process of deriving the adaptation-supporting self-* objectives is beyond the scope of this paper. Figure 4 depicts a 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

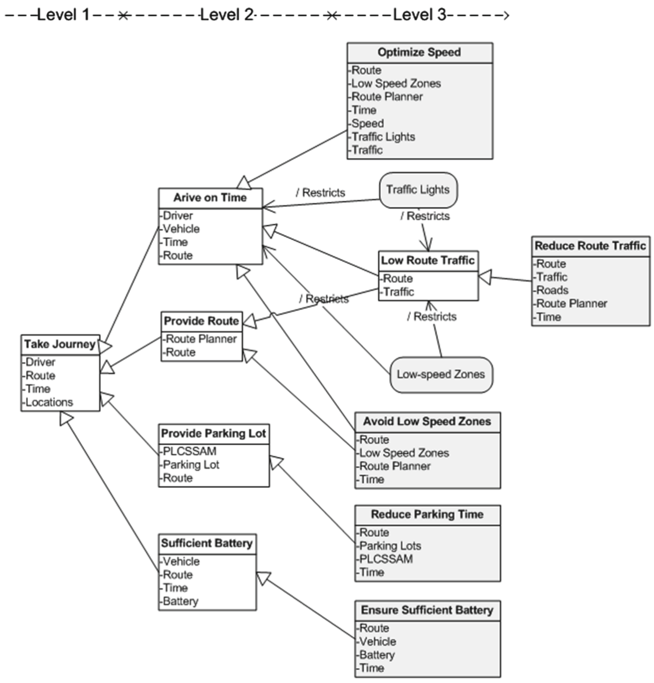

Fig. 4. eMobility Goals Model with Self-* Objectives for System Goals from Level 3

goals from the 3rd level. Basically, all the self-* objectives inherit the system goals they assist by providing behaviour alternatives with respect to these system goals. The eMobility system switches to one of the assisting self-* objectives when alternative autonomous behaviour is required (e.g., a vehicle needs to avoid low-speed zones). In addition, Fig. 4 depicts some of the environment constraints (e.g., "Traffic Lights" and "Low-speed Zones"), which may cause self-adaptation.

6.1 Specifying eMobility Ontology

In order to specify 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. 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.

Figure 5, 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

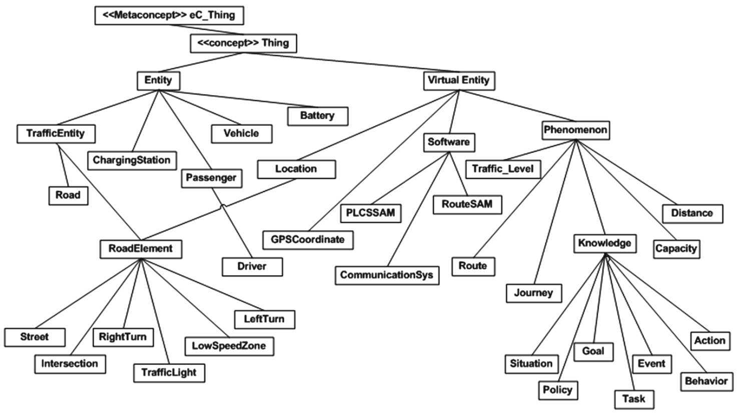

Fig. 5. eMobility Ontology Specified with KnowLang

concept is a TraficEntity and the Action concept is a Knowledge and consecutively Phenomenon, etc. Most of the concepts presented in Fig. 5 were derived from the eMobility Goals Model (see Fig. 4). Other concepts are considered as explicit and were derived from the KnowLang's specification model [22].

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 InSufficientBattery {/* to specify */}
    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 *Arrived OnTime* 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.

6.2 Specifying Self-Adaptive Behaviour

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 Fig. 5). Figure 6, depicts a concept tree representing the specified eMobility goals. Note that most of these goals were directly interpolated from the goals model (see Fig. 4).

Recall that KnowLang specifies goals as functions of states where any combination of states can be involved. A goal has an arriving state (Boolean function of states) and an optional departing state (another Boolean function of states). 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.

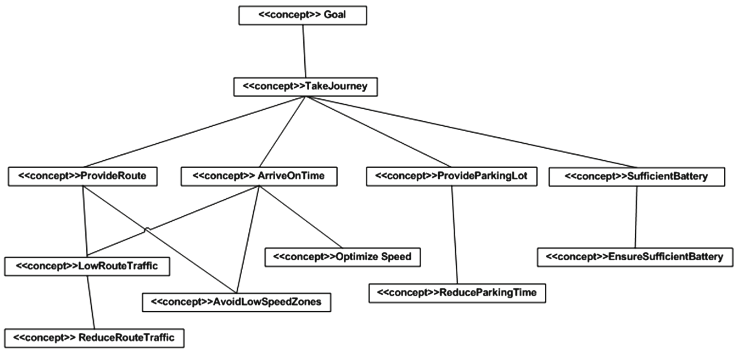
<<concept>> Goal
<<concept>>TakeJourney
<<concept>>ProvideRoute
<<concept>> ArriveOnTime
<<concept>>ProvideParkingLot
<<concept>>SufficientBattery
<<concept>>Optimize Speed
<<concept>>EnsureSufficientBattery
<<concept>>LowRouteTraffic
<<concept>>AvoidLowSpeedZones
<<concept>>ReduceParkingTime
<<concept>>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

Fig. 6. eMobility Ontology: eMobility Goal Concept Tree

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 $Reduce$ $RouteTraffic$ – 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 $recomputeRoads, takeAlternativeRoad$, 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 behaviour 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 [3] for probability assessment after action executions [22].

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 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, 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 behaviours or execution paths out of that situation. The following code represents the specification of the situation $RouteTrafficIncreased$, used for the specification of 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}
    }
  }
}
```

As shown, the situation is specified with $SITATION\_STATES$ (e.g., *InHigh* $Traffic$) and $SITUATION\_ACTIONS$ (e.g., *TakeAlterna-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-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: *TakeAlternative Road*.

Recall that situations are related to policies via relations. The following code demonstrates how we related the situation $RouteTrafficIncreased$ to the policy *Reduce-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* [22]. In our approach, we assume that eMobility sensors are controlled by software drivers (e.g., implemented in C++) where appropriate methods are

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

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 [22]:

– *RESOURCE* – measure resources like capacity;
– *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 concept (see Sect. 6.1).

```
// 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 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 }
  }
}
```

7 Conclusion and Future Work

In the course of this R&D process, we shaped our research activities towards focusing on the KnowLang Framework where our ultimate goal is to structure computerized knowledge so that a computerized system can effectively process it and gain awareness capabilities and eventually derive its own behaviour. To provide comprehensive and powerful specification formalism, we developed a powerful multi-tier specification model where ontologies are integrated with rules and Bayesian networks. The approach allows for efficient and comprehensive knowledge structuring and awareness based on logical and statistical reasoning. We used the KnowLang notation to specify some knowledge models for different case studies. This exercise demonstrated the ability of KnowLang to handle KR for systems from different application domains. A very important feature is the KnowLang mechanism for self-adaptive behaviour where knowledge representation and reasoning help to establish the vital connection between knowledge, perception, and actions realizing self-adaptive behaviour. The knowledge is used against the perception of the world to generate appropriate actions in compliance to some goals and beliefs.

Future work is mainly concerned with further development of the KnowLang Reasoner as part of the full implementation of the KnowLang Framework, involving tools and a test bed for verification and validation of KnowLang models.

Acknowledgements. This work was supported, in part, by Science Foundation Ireland grant 13/RC/ 2094_P2 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero–the Science Foundation Ireland Research Centre for Software (www.lero.ie) and by University of Limerick, Limerick, Ireland.

References

1. Baader, F., Nutt, W.: Basic description logics. In: Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.) The Description Logic Handbook, pp. 43–95. Cambridge University Press, Cambridge (2003)
2. Brachman, R., Levesque, H.: Knowledge Representation and Reasoning. Elsevier, New York (2004)
3. Ewens, W., Grant, G.: Stochastic Processes (i): Poison Processes and Markov Chains. Statistical Methods in Bioinformatics, 2nd edn. (2005)
4. Halpern, J.Y.: An analysis of first-order logics of probability. Artif. Intell. **46**, 311–350 (1990)
5. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
6. Knuth, D.E.: Backus normal form vs. backus naur form. Commun. ACM **7**(12), 735–736 (1964)
7. Neapolitan, R.: Learning Bayesian Networks. Prentice Hall, Hoboken (2003)
8. Serbedzija, N., et al.: D7.3: Third Report on WP7 Integration and Simulation Report for the ASCENS Case Studies (2013). ASCENS Deliverable
9. Serbedzija, N., et al.: D7.2: Second Report on WP7 Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility (2012). ASCENS Deliverable
10. Serbedzija, N., et al.: D7.1: First Report on WP7 Requirement Specification and Scenario Description of the ASCENS Case Studies (2011). ASCENS Deliverable
11. Vassev, E.: ASSL: Autonomic System Specification Language - A Framework for Specification and Code Generation of Autonomic Systems. LAP Lambert Academic Publishing (2009)
12. Vassev, E.: KnowLang grammar in BNF. Technical report. Lero-TR-2012-04, Lero, University of Limerick, Ireland (2012)
13. Vassev, E., Hinchey, M.: The challenge of developing autonomic systems. IEEE Comput. **43**(12), 93–96 (2010)

14. Vassev, E., Hinchey, M.: Towards a formal language for knowledge representation in autonomic service-component ensembles. In: Proceedings of the 3rd International Conference on Data Mining and Intelligent Information Technology Applications (ICMIA 2011), pp. 228–235. AICIT, IEEE Xplore (2011)
15. Vassev, E., Hinchey, M.: Awareness in software-intensive systems. IEEE Comput. **45**(12), 84–87 (2012)
16. Vassev, E., Hinchey, M.: Efficient reasoning with ambient trees for space exploration. In: Vinh, P.C., Hung, N.M., Tung, N.T., Suzuki, J. (eds.) ICCASA 2012. LNICST, vol. 109, pp. 176–182. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36642-0_18
17. Vassev, E., Hinchey, M.: Knowledge representation for cognitive robotic systems. In: Proceedings of the 15th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing Workshops (ISCORCW 2012), pp. 156–163. IEEE Computer Society (2012)
18. Vassev, E., Hinchey, M.: Knowledge representation with KnowLang - the marXbot case study. In: Proceedings of the 11th IEEE International Conference on Cybernetic Intelligent Systems (CIS 2012). IEEE Computer Society (2012)
19. Vassev, E., Hinchey, M.: Knowledge representation and reasoning for self-adaptive behavior and awareness. TCCI - Special Issue on ICECCS 2012 (2013, pending)
20. Vassev, E., Hinchey, M., Gaudin, B.: Knowledge representation for self-adaptive behavior. In: Proceedings of C* Conference on Computer Science & Software Engineering (C3S2E 2012), pp. 113–117. ACM (2012)
21. Vassev, E., Hinchey, M., Gaudin, B., Nixon, P., Bicocchi, N., Zambonelli, F.: D3.1: First Report on WP3. Software requirements, knowledge modeling and knowledge representation for self-awareness - report and survey with experimental results for intelligent multi-agent systems (2011). ASCENS Deliverable
22. Vassev, E., Hinchey, M., Montanari, U., Bicocchi, N., Zambonelli, F., Wirsing, M.: D3.2: Second Report on WP3: The KnowLang Framework for Knowledge Modeling for SCE Systems (2012). ASCENS Deliverable