Knowledge-Based Self-Adaptation

Emil Vassev
Lero, University of Limerick
Limerick, Ireland
emil.vassev@lero.ie

Mike Hinchey
Lero, University of Limerick
Limerick, Ireland
mike.hinchey@lero.ie

Abstract

Self-adaptation is the ability of a system to adapt its behavior and/or computational structures to changes in the execution environment. The paradigm requires that the system engages in various interactions where important structural and dynamic aspects of the environment are perceived. In this paper, we present an approach to implementing self-adaptation capabilities with KnowLang, a special framework for knowledge representation and reasoning. KnowLang provides for a special knowledge context and a special reasoner operating in that context. The reasoner communicates with the host system via special ASK and TELL operators allowing for knowledge queries and updates. Whereas TELL Operators feed the knowledge context with important information driven by errors, executed actions, new sensory data, etc., ASK Operators 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.

1. Introduction

Developing intelligent systems with Knowledge Representation and Reasoning (KR&R) has been an increasingly interesting topic for years. Examples are found in semantic mapping [3], improving planning and control aspects [6], and most notably HRI systems [5, 4]. Overall, KR&R strives to solve complex problems where the operational environment is non-deterministic and a system needs to reason at runtime to find missing answers. Decision-making is a complex process that is often based on more than logical conclusions. Probability and statistics may provide for the so-called probabilistic and statistical reasoning intended to capture uncertain knowledge in which additive probabilities are used to represent degrees of belief of rational agents in the truth of statements. For example, the purpose of a statistical inference might be to draw conclusions about a population based on data obtained from a sample of that population. Probability theory and Baye’s theorem [8] lay the basis for such reasoning where Bayesian networks [7] are used to represent belief probability distributions, which actually summarize a potentially infinite set of possible circumstances. The key point is that nodes in a Bayesian network have direct influence on other nodes and given values for some nodes, it is possible to infer the probability distribution for values of other nodes. How a node influences another node is defined by the conditional probability for the nodes, usually based on past experience. The experience can be associated with the success of the actions generated in the physical environment by the intelligent system. Maintaining an execution history of the actions shall help that system eventually compute the success probability for those actions. In that way, the system may learn (infer new knowledge) not to execute actions that traditionally have low success rate.

In this paper, we present KnowLang, an approach to knowledge representation for self-adaptive behavior and awareness based on the methodology discussed above. KnowLang [12, 13] is an initiative undertaken by Lero - the Irish Software Engineering Research Center within Lero’s mandate in the ASCENS Project [1]. Autonomic Service-Component ENsembles (ASCENS) is an FP7 (Seventh Framework Program) project targeting the development of a coherent and integrated set of methods and tools providing a comprehensive development approach to developing ensembles (or swarms) of intelligent, self-aware and adaptive service components. One of the main scientific contributions that we expect to achieve with ASCENS is related to KR&R. Note that it is of major importance for an ASCENS system to acquire and structure comprehensive knowledge in such a way that it can be effectively and efficiently processed, so such a system becomes aware of itself and its environment. Moreover, ASCENS is an AI project tackling self-adaptation of systems operating in open-ended environment, e.g., our physical world. Such systems need to be developed with initial knowledge and learning capabilities based on knowledge processing and awareness. It is very important how the system knowledge is both structured and
modeled to provide essence of self-adaptation.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 presents KnowLang as a formal specification language for knowledge representation in self-adaptive systems. Section 4 presents the KnowLang Reasoner by emphasising the operational semantics of special ASK and TELL operators used by the decision-making process to derive self-adaptive behavior. In Section 5, we present a proof-of-concept case study. Finally, Section 6 presents a brief conclusion and future work.

2. Related Work

Knowledge representation for self-adaptive systems is a wide open research area with only a limited number of approaches yet considered. The work that is most similar in spirit to our own is that on developing cognitive robots relying on the so-called deliberative controllers. Architectures for autonomous control in robotic systems require concurrent embedded real-time performance, and are typically too complex to be developed and operated using conventional programming techniques. The core of an autonomous controller is an execution system that executes commands and monitors the environment [2]. Execution systems with deliberative controllers are based on knowledge that contains an explicitly represented symbolic model of the world. Deliberation is the explicit consideration of alternative behaviors (courses of actions).

In [14] an agent programming language called Goal is used to program a cognitive robot control architecture that combines low-level sub-symbolic control with high-level symbolic control. The Goal language helps to realize a cognitive layer whereas low-level execution control and processing of sensor data are delegated to components in other layers. Similar to KnowLang, Goal supports a goal-oriented behavior and decomposition of complex behavior by means of modules that can focus their attention on relevant subgoals. However, KnowLang is far more expressive than Goal, especially at the level of modeling self-adaptive behavior, which is not supported by Goal. The integration of situations, goals, policies, and actions with a Bayesian network probability distribution allows for self-adaptation based on both logical and statistical reasoning.

In [9] the high-level language Golog is used for robot programming. Golog supports writing control programs in a high-level logical language, and provides an interpreter that, given a logical axiomatization of a domain, will determine a plan. Similar to KnowLang, Golog also supports actions and situations (actually the language incorporates Situation Calculus), but again, KnowLang is far more expressive with its Ontology-logical framework knowledge structuring. Moreover, Golog does not provide a means for self-adaptive KR, which is provided by KnowLang.

3. KnowLang

A key feature of KnowLang is a formal language with a multi-tier knowledge specification model allowing for integration of ontologies together with rules and Bayesian networks [7]. The language aims at efficient and comprehensive knowledge structuring and awareness based on logical and statistical reasoning. It helps us tackle [12]: 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) [12] and knowledge representation for autonomic behavior [13]. By applying the KnowLang’s multi-tier specification model (see Figure 1) we build a Knowledge Base (KB) structured in three main tiers [12]: 1) Knowledge Corpuses; 2) KB Operators; and 3) Inference Primitives. The tier of Knowledge Corpuses is used to specify KR structures. The tier of KB Operators provide access to Knowledge Corpuses via special classes of ASK and TELL Operators where ASK Operators are dedicated to knowledge querying and retrieval and TELL Operators allow for knowledge update. When we specify knowledge with KnowLang, we build a KB with a variety of knowledge structures such as ontologies, facts, rules and constraints where we need to specify the ontologies first in order to provide the “vocabulary” for the other knowledge structures. A KnowLang ontology is specified over concept trees, object trees, relations and predicates. Each concept is specified with special properties and functionalities.

Figure 1. KnowLang Specification Model
and is hierarchically linked to other concepts through \textit{PARENTS} and \textit{CHILDREN} relationships. For reasoning purposes every concept specified with KnowLang has an intrinsic \textit{STATE} attribute that may be associated with a set of possible \textit{state values} the concept instances may be in. The concept instances are considered as objects and are structured in object trees - a conceptualization of how objects existing in the world of interest are related to each other. The relationships in an object tree are based on the principle that objects have properties, where the value of a property is another object, which in turn also has properties. Moreover, concepts and objects might be connected via \textit{relations}. Relations are binary and may have probability-distribution attribute (e.g., over time, over situations, over concepts’ properties, etc.). Probability distribution is provided to support probabilistic reasoning and by specifying relations with probability distributions we actually specify Bayesian networks connecting the concepts and objects of an ontology. Figure 2 shows a KnowLang specification sample demonstrating both the language syntax [10] and its visual counterpart - a concept map based on interrelations with no probability distributions. Modeling knowledge with KnowLang requires a few phases:

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

3.1 Modeling Self-adaptive Behavior

KnowLang employs special knowledge structures and a reasoning mechanism for modeling autonomic self-adaptive behavior [13]. Such a behavior can be expressed via KnowLang policies, events, actions, situations and relations between policies and situations (see Definitions 1 through 10). Policies (Π) are at the core of autonomic behavior. A policy \( \pi \) has a goal \( g \), policy situations \( Si_\pi \), policy-situation relations \( R_\pi \), and policy conditions \( N_\pi \) mapped to policy actions \( A_\pi \) where the evaluation of \( N_\pi \) may eventually (with some degree of probability) imply the evaluation of actions (denoted with \( N_\pi \xrightarrow{\pi} A_\pi \)) (see Definition 2). A condition is a Boolean expression over ontology (see Definition 4), e.g., the occurrence of a certain event.

\textit{Policy situations} \( Si_\pi \) are situations (see Definition 7) that may trigger (or imply) a policy \( \pi \), in compliance with the policy-situations relations \( R_\pi \) (denoted with \( Si_\pi \xrightarrow{R_\pi} \pi \)), thus implying the evaluation of the policy conditions \( N_\pi \) (denoted with \( \pi \rightarrow N_\pi \)) (see Definition 2). Therefore, the optional policy-situation relations \( R_\pi \) justify the relationships between a policy and the associated situations (see Definition 10). Note that in order to allow for self-adaptive behavior, \textit{relations} must be specified to connect policies with situations over an optional probability distribution \( Z \) where a policy might be related to multiple situations and vice versa. Probability distribution is provided to support probabilistic reasoning and to help the reasoner choose the most probable situation-policy “pair”. Thus, we may specify a few relations connecting a specific situation to different policies to be undertaken when the system is in that particular situation and the probability distribution over these relations (involving the same situation) should help the reasoner decide which policy to choose (denoted with \( \pi \rightarrow Z - \text{π} \) - see Definition 10). Hence, the presence of \textit{probabilistic beliefs} at both mappings and policy relations justifies the probability of policy execution, which may vary with time. A goal \( g \) is a desirable transition to a state or from a specific state to another state (denoted 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 \( (A_{\pi}\) (actions executed to get to state \( s \)), actions \( A_\pi \) that can be performed from state \( s \) and an optional history of events \( E_\pi \) that eventually occurred to get to state \( s \) (see Definition 8).

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

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

\( A_\pi \subset A, N_\pi \xrightarrow{Z} A_\pi \quad (A_\pi - \text{Policy Actions}) \)

\( Si_\pi \subset Si, Si_\pi \xrightarrow{R_\pi} \pi \rightarrow N_\pi \quad (Si_\pi - \text{Policy Sits}) \)

\( R_\pi \subset R \quad (R_\pi - \text{Policy-Situation Relations}) \)

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

\textbf{Def. 4} \( n := be(O) \) \quad (Boolean Expression over Ontology)

\textbf{Def. 5} \( g := (\Rightarrow s')|(s \Rightarrow s') \) \quad (Goal)

\textbf{Def. 6} \( s := be(O) \) \quad (State)

\textbf{Def. 7} \( Si := \{si_1, si_2, \ldots, si_n\}, n \geq 0 \) \quad (States)

\textbf{Def. 8} \( si := <s, A_{\pi}, E_{\pi}, A_{si} > \) \quad (Situation)

\( A_{\pi} \subset A \quad (A_{\pi} - \text{Possible Actions}) \)

\( A_{si} \subset A \quad (A_{si} - \text{Executed Actions}) \)

\( E_{\pi} \subset E \quad (E_{\pi} - \text{Situation Events}) \)

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

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

\( si \in Si, \pi \in \Pi, si \xrightarrow{Z} \pi \)
Ideally, KnowLang policies are specified to handle specific situations, which may trigger the application of policies. A policy exhibits a behavior via actions generated in the environment or in the system itself. Specific conditions determine, which specific actions (among the actions associated with that policy - see Definition 2) shall be executed. These conditions are often generic and may differ from the situations triggering the policy. Thus, the behavior not only depends on the specific situations a policy is specified to handle, but also depends on additional conditions. Such conditions might be organized in a way allowing for synchronization of different situations on the same policy. When a policy is applied, it checks what particular conditions are met and performs the mapped actions (see map(\(N_\pi, A_\pi, [Z]\)) - see Definition 2). An optional probability distribution may 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.

3.2 Converting Sensory Data to KR

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. 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 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., 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 [10]:

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

4. 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 a special KnowLang Reasoner. The reasoner communicates with the system and operates in the KR Context, a context formed by the represented knowledge (see Figure 3).
The KnowLang Reasoner should be supplied as a component hosted by the system and thus, it runs in the system’s Operational Context as any other system’s component. However, it operates in the KR Context and on the KR symbols (represented knowledge). The system talks to the reasoner via special ASK and TELL Operators allowing for knowledge queries and knowledge updates (See Figure 3). Upon demand, the KnowLang Reasoner can also build up and return a self-adaptive behavior model - a chain of actions to be realized in the environment or in the system.

4.1 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 [11]:

- `TELL_ERR` - tells about a raised error;
- `TELL_SENSOR` - tells about new data collected by a sensor;
- `TELL_ACTION` - tells about action execution;
- `TELL_ACTION` (behavior) - tells about action execution as part of behavior performance;
- `TELL_OBJ_UPDATE` - tells about a possible object update;
- `TELL_CNCEPT_UPDATE` - tells about a possible concept update;
- `ASK_BEHAVIOR` - asks for self-adaptive behavior;
- `ASK_BEHAVIOR(goal)` - asks for self-adaptive behavior to achieve certain goal;
- `ASK_BEHAVIOR(situation, goal)` - asks for self-adaptive behavior to achieve certain goal when departing from a specific situation;
- `ASK_BEHAVIOR(state)` - asks for self-adaptive behavior to go to a certain state;
- `ASK_RULE_BEHAVIOR(conditions)` - asks for rule-based behavior;
- `ASK_CURR_STATE(object)` - asks for the current state of an object;
- `ASK_CURR_STATE` - asks for the current system state;
- `ASK_CURR_SITUATION` - asks for the current situation.

The following two subsections provide a brief presentation of the operational semantics [11] of two KB Operators.

4.2 The TELL_SENSOR Operator

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

(1) \( \frac{\sigma \rightarrow \text{tell_sensor}(d,s)}{\sigma' \rightarrow \text{findMetricConcept}(c,s) \rightarrow \langle c, \sigma' \rangle} \)

(2) \( \frac{\sigma \rightarrow \text{tell_sensor}(d,s)}{\sigma' \rightarrow \text{findMetricObject}(c,s) \rightarrow \langle c, \sigma' \rangle} \)

(3) \( \frac{\sigma \rightarrow \text{tell_sensor}(d,s)}{\sigma' \rightarrow \text{findMetricObject}(c,s) \rightarrow \langle a_m, \sigma' \rangle} \)

(4) \( \frac{\sigma \rightarrow \text{findCurrentSituation}(\sigma_m), \sigma \rightarrow \langle E_m, \sigma' \rangle}{\sigma' \rightarrow \text{o_c-fired event}} \)

As shown in Rule 1, the call of the \text{tell_sensor()} function (a method implementing the system call of the TELL_SENSOR Operator) triggers a context switching \( \sigma \rightarrow \sigma' \), i.e., the process control is passed to the KnowLang Reasoner, which operates in the KRC only. Further, this context switching initiates an internal for KRC call of the TELL_SENSOR Operator, which triggers the retrieval of the metric concept specified in the KB to represent the sensor’s class implemented in the program (see Section 3.2). The \text{findMetricConcept(s)} function is used to denote the execution of a traversal algorithm that finds a metric concept by an implementation reference string (s carries information about the sensor implementation, e.g., class). Then, if the metric concept has been successfully found, the reasoner looks up the concept instance representing the sensor’s object in the program implementation (denoted in Rule 2 with the \text{findMetricObject(c,s)} function). If the concept instance is successfully found, then the reasoner updates that instance accordingly (denoted in Rule 3 with the \text{update(a_m,d)} function). Next, the reasoner looks up and fires all the events specified to be activated by a change in this specific metric (see Rule 4 and the abstract functions \text{findMetricEvents(a_m)} and \text{fireEvent(e_m)} respectively). Note that KnowLang events can be specified to be activated by a data change in specific metrics. The following fragment of the KnowLang Grammar [10] demonstrates that:

\text{State-Body} \rightarrow \text{Bln-Expr}

\text{Bin-Reln} \rightarrow \text{OCCURRED ( Event-Name )}

When an event is fired, the reasoner creates a new event instance (event object) in the KRC. Further, the reasoner looks up all the concept instances whose states depend on the existence of that event (see Rule 5 and abstract function \text{findDependedObjects(e)}). Recall that states in KnowLang are expressed as Boolean expression over ontology (see Section 3.1). The occurrence of an event can be used within such expressions and thus, events can be used to specify states. The following are fragments of the KnowLang Grammar [10] demonstrating that:

4.3 The \text{ASK_BEHAVIOR} Operator

\text{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 \text{ASK_BEHAVIOR} Operator - σ states for OC and σ' states for KRC. For clarity reasons, we do not show the change in KRC after updates have been made in that context.

(8) \( \frac{\sigma \rightarrow \text{askBehavior}()} {\sigma' \rightarrow \text{askBehavior}()} \)

(9) \( \frac{\sigma \rightarrow \text{askBehavior}()} {\sigma' \rightarrow \text{askBehavior}()} \)

(10) \( \frac{\sigma \rightarrow \text{applyPolicy}(\pi), \sigma' \rightarrow \text{applyPolicy}(\pi)} {\sigma' \rightarrow \text{applyPolicy}(\pi)} \)

(11) \( \frac{\sigma \rightarrow \text{applyPolicy}(\pi), \sigma' \rightarrow \text{applyPolicy}(\pi)} {\sigma' \rightarrow \text{applyPolicy}(\pi)} \)

(12) \( \frac{\sigma \rightarrow \text{applyPolicy}(\pi), \sigma' \rightarrow \text{applyPolicy}(\pi)} {\sigma' \rightarrow \text{applyPolicy}(\pi)} \)

(13) \( \frac{\sigma \rightarrow \text{applyPolicy}(\pi), \sigma' \rightarrow \text{applyPolicy}(\pi)} {\sigma' \rightarrow \text{applyPolicy}(\pi)} \)
As shown in Rule 8, to ask for behavior, the system calls the \textit{ask\_behavior} function (a method implementing the system call of the ASK\_BEHAVIOR Operator), which triggers a context switching $\sigma \xrightarrow{\text{ask\_behavior}()} \sigma'$. 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 \textit{findCurrentSituation()} abstract function) to find the situation the system is currently in. The current situation will be approximately determined based on the \textit{global system state}. Once the current situation is successfully determined (see the second premise in Rule 9), the reasoner needs to find all the policies associated with that situation. Thus, the reasoner looks up all the \textit{situation-policy relations} the current situation participates in (denoted with the \textit{findSitnPolcyRltns(si)} - see the conclusion in Rule 9). Next, the relation with the \textit{highest probability rate} is selected (recall that KnowLang Relations may be associated with a probability rate - see Definition 10 in Section 3.1), which helps to determine the \textit{most appropriate policy} for that particular situation (see the conclusion in Rule 10). The selected policy is applied (see Rule 11). The evaluation of a policy triggers a \textit{mapping operation} where any \textit{policy condition} that is held (the conditions are Boolean expressions) is mapped to appropriate actions with eventual \textit{probability rate} (see Definition 2 in Section 3.1). This operation selects pairs “actions subset”-“probability rate” (see the conclusion in Rule 12). Next, the reasoner selects from these pairs the one with the highest probability rate to extract the \textit{subset of actions} to be executed (see the last premise and conclusion in Rule 13). The extracted subset of possible actions has to be recorded as a \textit{behavior model} (see the conclusion in Rule 14 where this is denoted with the \textit{recordBehavior}($\pi_{si}, A_{si}'$, $\sigma'$) abstract function). Finally, the KnowLang Reasoner returns the recorded behavior model to the system with a context switching back to OC (see Rule 15). Note that the behavior model must comprise only actions allowed to be executed from the actual situation (see Definition 8 in Section 3.1).

5. Case Study

To illustrate \textit{autonomic self-adaptive behavior} based on this approach, we are going to elaborate on a “trapped robot case study” by assuming that a \textit{trapped robot} keeps sending a help signal and another robot (called Robot\_A) is receiving that signal. Eventually, the \textit{sensory data} representing the received signal will be passed to the Robot’s KB via system calls of TELL Operators. Then, the system may call an ASK\_BEHAVIOR operator to get the most appropriate behavior in the current situation. Let us assume that we have used KnowLang to specify a KB for Robot\_A where in addition to another explicit knowledge, we have also specified policy $\pi_1$ (see Figure 4). Although we are missing the basic specification of the involved actions, goal, situation and relation, we can conclude that the current situation $s_{11}$: “a robot needs assistance” will trigger a policy $\pi_1$: “go to the signal source” if the relation $r_1(s_{11}, \pi_1)$ has the higher probabilistic belief rate. The $\pi_1$ policy will realize actions \textit{Turn} and \textit{Move} iff the robot’s battery is charged at least 50% and there is no other higher priority task to finish up first (currently ongoing or scheduled). The ASK\_BEHAVIOR Operator will return the generated behavior as a sequence of actions, e.g., {\textit{Action.Turn} (\textit{Action.GetSignalAngle}), \textit{Action.Move}}.

Next, Robot\_A will perform the generated actions and
will start moving towards the signal. Let us assume that while moving, at certain point, Robot_A will hit a wall and get into a situation \(si_2\): "road is blocked", which by specification is related to policy \(\pi_2\): "avoid obstacle" (see Figure 4). Policy \(\pi_2\) will force the robot to turn right and move, because of the initial probability distribution in the MAPPING sections. Eventually, Robot_A will reach a hole in the wall and thus, will accomplish the \(\pi_2\)'s goal \(g_2\): "free road". Then it will go back to the initial situation \(si_1\): "a robot needs assistance", which will trigger the policy \(\pi_1\): "go to the signal source" and the robot will start moving again towards the trapped robot. Let us suppose that there are more walls on the route to the trapped robot and any time when Robot_A gets into situation \(si_2\): "road is blocked" it will continue applying the \(\pi_2\) policy by avoiding the wall from the right side until it hits a very long wall on the right side and gets into a situation \(si_3\): "signal is lost". This new situation shall trigger another policy \(\pi_3\): "go back until signal appears", which will move the robot back to a point where the help signal appears again and then, the robot will get back to situation \(si_2\) and policy \(\pi_2\). Following \(\pi_2\), the robot can fall again into \(si_3\) and then back to \(si_2\). However, every time when policy \(\pi_2\) fails to accomplish its goal \(g_2\): "free road", the KnowLang Reasoner re-computes the probability distribution in the MAPPING sections, which eventually may lead to a point where by applying policy \(\pi_2\) the robot will turn left and move, i.e., it will self-adapt to the current situation and will try to avoid the wall from the left side.

6. Conclusion and Future Work

In this paper, we have presented the KnowLang Framework as an approach to KR&R allowing for self-adaptive behavior in software-intensive systems. The ultimate goal is to structure computerized knowledge so that a computerized system can effectively process it and gain awareness capabilities and eventually derive its own behavior. The approach allows for efficient and comprehensive knowledge structuring and awareness based on logical and statistical reasoning. The KnowLang Reasoner provides for a mechanism for self-adaptive behavior where KR&R help to establish the vital connection between knowledge, perception, and actions realizing self-adaptive behavior. The knowledge is used against the perception of the world to generate appropriate actions in compliance to some goals and beliefs. The mechanism incorporates special ASK and TELL operators used by the system to talk to the KnowLang Reasoner.

Note that KnowLang is still under development as part of the ASCENS international European project. Our plans are to completely develop KnowLang including a toolset for formal validation. Once fully implemented, KnowLang will be used to specify knowledge representation and autonomic behavior in different case studies.

Acknowledgments

This work was supported by the European Union FP7 Integrated Project Autonomic Service-Component Ensembles (ASCENS) and by Science Foundation Ireland grant 03/CE2/I303_1 to Lerothe Irish Software Engineering Research Centre at University of Limerick, Ireland.

References