Modeling Swarm Robotics with KnowLang

Emil Vassev and Mike Hinchey

Lero–the Irish Software Engineering Research Centre,
University of Limerick, Limerick, Ireland
{emil.vassev,mike.hinchey}@lero.ie

Abstract. Swarm robotics has emerged as a paradigm whereby intelligent agents are considered to be autonomous entities that interact either cooperatively or non-cooperatively. The concept is biologically-inspired and offers many advantages compared with single-agent systems, such as: greater redundancy, reduced costs and risks, and the ability to distribute the overall work among swarm members, which may result in greater efficiency and performance. The distributed and local nature of these systems is the main factor in the high degree of parallelism displayed by their dynamics that often results in adaptation to changing environmental conditions and robustness to failure. This paper presents a formal approach to modeling self-adaptive behavior for swarm robotics. The approach relies on the KnowLang language, a formal language dedicated to knowledge representation for self-adaptive systems.

1 Introduction

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

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

In this paper, we present a formal approach to modeling self-adaptive behavior of swarm robotics. We use KnowLang, a formal framework under development

P.C. Vinh et al. (Eds.): ICTCC 2014, LNICST 144, pp. 13–22, 2015.
DOI: 10.1007/978-3-319-15392-6_2

under the mandate of the FP7 project, ASCENS [1]. KnowLang's notation is a formal language dedicated to knowledge representation for self-adaptive systems, so the framework provides both a notation and reasoning to deal with self-adaptation.

The rest of this paper is organized as follows. Section 2 presents the ARE approach that helps us capture the requirements for self-adaptive behavior. Section 3 describes our swarm robotics case study. Section 4 presents our approach to specifying the self-adaptive behavior of swarm robots with KnowLang. Finally, Section 5 provides brief concluding remarks and a summary of our future goals.

2 Requirements for Self-adaptive Behavior

We aim to capture self-adaptive behavior so that it can be properly designed and subsequently implemented. To do so, we consider that self-adaptive behavior extends the regular objectives of a system upstream with special self-managing objectives, also called self-* objectives [6]. Basically, the self-* objectives provide autonomy features in the form of a system's ability to automatically discover, diagnose, and cope with various problems. This ability depends on the system's degree of autonomicity, quality and quantity of knowledge, awareness and monitoring capabilities, and quality characteristics such as adaptability, dynamicity, robustness, resilience, and mobility. The approach for capturing all of these requirements is called Autonomy Requirements Engineering (ARE) [4–6]. This approach aims to provide a complete and comprehensive solution to the problem of autonomy requirements elicitation and specification. Note that the approach exclusively targets the self-* objectives as the only means to explicitly determine and define autonomy requirements. Thus, it is not meant to handle the regular functional and non-functional requirements of the systems, presuming that those might by tackled by the traditional requirements engineering approaches, e.g., use case modeling, domain modeling, constraints modeling, etc. Hence, functional and non-functional requirements may be captured by the ARE approach only as part of the self-* objectives elicitation.

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

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

supportive and alternative self-* objectives along with the necessary capabilities and quality characteristics.

Finally, the last step after defining the autonomy requirements per the system's objectives is the formalization of these requirements, which can be considered as a form of formal specification or requirements recording. The formal notation used to specify the autonomy requirements is KnowLang [7]. The process of requirements specification with KnowLang extends over a few phases:

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

To specify self-* objectives with KnowLang, we use special policies associated with goals, special situations, actions (eventually identified as system capabilities), metrics, etc.[7]. Hence, self-* objectives are represented as policies describing at an abstract level what the system will do when particular situations arise. The situations are meant to represent the conditions needed to be met in order for the system to switch to a self-* objective while pursuing a system goal. Note that policies rely on actions that are *a priori* defined as functions of the system. In the case that such functions have not been defined yet, the needed functions should be considered as autonomous functions and their implementation will be justified by the ARE's selected self-* objectives.

According to the KnowLang semantics, in order to achieve specified goals (objectives), we need to specify policy-triggering *actions* that will eventually change the system states, so the desired ones, required by the goals, will become effective [7]. Note that KnowLang policies allow the specification of autonomic behavior (autonomic behavior can be associated with self-* objectives), and therefore, we need to specify at least one policy per single goal; i.e., a policy that will provide the necessary behavior to achieve that goal. Of course, we may specify multiple policies handling same goal (objective), which is often the case with the self-* objectives and let the system decide which policy to apply taking into consideration the current situation and conditions. The following is a formal presentation of a KnowLang policy specification [7].

Policies (Π) are at the core of autonomic behavior (autonomic behavior can be associated with autonomy requirements). A policy π has a goal (g), policy situations (Si_π), policy-situation relations (R_π), and policy conditions (N_π) mapped to policy actions (A_π) where the evaluation of N_π may eventually (with some degree of probability) imply the evaluation of actions (denoted with $N_\pi \overset{[Z]}{\rightarrow} A_\pi$ (see Definition 2). A condition is a Boolean function over ontology (see Definition 4), e.g., the occurrence of a certain event.

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

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

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

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

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

Definition 6. $s := be(O)$ (State)

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

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

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

A goal g is a desirable transition to a state or from a specific state to another state (denoted with $s \Rightarrow s'$) (see Definition 5). A state s is a Boolean expression over ontology ($be(O)$)(see Definition 6), e.g., "a specific property of an object must hold a specific value". A situation is expressed with a state (s), a history of actions ($A \overset{\leftarrow}{si}$) (actions executed to get to state s), actions A_{si} that can be performed from state s and an optional history of events $E \overset{\leftarrow}{si}$ that eventually occurred to get to state s (see Definition 8).

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

3 The Ensemble of Robots Case Study

The ensemble of robots case study targets swarms of intelligent robots with self-awareness capabilities that help the entire swarm acquire the capacity to reason, plan, and autonomously act. The case study relies on the marXbot robotics platform [2], which is a modular research robot equipped with a set of devices that help the robot interact with other robots of the swarm or the robotic environment. The environment is defined as an arena where special cuboid-shaped obstacles are present in arbitrary positions and orientations. Moreover, the environment may contain a number of light sources, usually placed behind the goal area, which act as environmental cues used as shared reference frames among all robots.

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

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

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

For the purpose of the Ensemble of Robots case study, we developed a simple scenario that requires self-adaptive behavior of the individual marXbot robots

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

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

4 Formalizing Swarm Robotics with KnowLang

Following the scenario described in Section 3, we applied the ARE approach (see Section 2) and derived the 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 along with the process of deriving the adaptation-supporting self-* objectives is beyond the scope of this paper. These will be addressed in a future paper.

Based on the rationale above, we applied the ARE approach (see Section 2) and derived the system's 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 swarm robotics along with the process of deriving the adaptation-supporting self-* objectives is beyond the scope of this paper.

Figure 1 depicts the ARE goals model for swarm robotics where goals are organized hierarchically at three different levels. As shown, the goals from the first two levels (e.g., "Rescue Victims", "Protect against Radiation", and "Move Victims away") are *main system goals* captured at different levels of abstraction. The 3rd level is resided by *self-* objectives* (e.g., "Clean Debris", "Optimize Rescue Operation", and "Avoid Radiation Zones") and *supportive goals* (e.g., "Exploration and Mapping" and "Find Victim") associated with and assisting the 2nd-level goals. Basically, all self-* objectives inherit the system goals they assist by providing behavior alternatives with respect to these system goals. The system switches to one of the assisting self-* objectives when alternative autonomous behavior is required (e.g., a robot needs to avoid a radiation zone). In addition, Figure 1 depicts some of the environmental constraints (e.g., "Radiation" and "Debris"), which may cause self-adaptation.

In order to specify the autonomy requirements for swarm robotics, the first step is to specify a knowledge base (KB) representing the swarm robotics system in question, i.e., robots, victims, radiation, debris, etc. To do so, we need

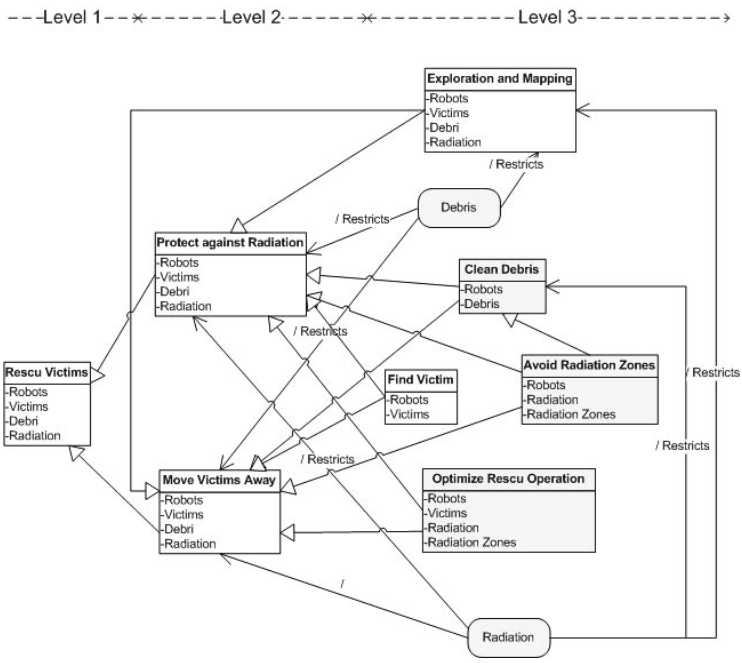

Fig. 1. Swarm Robotics Goals Model with Self-* Objectives

to specify ontology structuring the knowledge domain of the case study. Note that this domain is described via domain-relevant concepts and objects (concept instances) related through relations. To handle explicit concepts like situations, goals, and policies, we grant some of the domain concepts with explicit state expressions where a state expression is a Boolean expression over the ontology (see Definition 6 in Section 2). Note that being part of the autonomy requirements, knowledge plays a very important role in the expression of all the autonomy requirements (see Section 2).

Figure 2, depicts a graphical representation of the swarm robotics ontology relating most of the domain concepts within a swarm robotics system. Note that the relationships within a concept tree are "is-a" (inheritance), e.g., the *Radiation_Zone* concept is an *EnvironmentEntity* and the *Action* concept is a *Knowledge* and consecutively *Phenomenon*, etc. Most of the concepts presented in Figure 2 were derived from the Swarm Robotics Goals Model (see Figure 1). Other concepts are considered *explicit* and were derived from the KnowLang specification model [8].

The following is a sample of the KnowLang specification representing the *Robot* concept. As specified, the concept has properties of other concepts, functionalities (actions associated with that concept), states (Boolean expressions validating a specific state), etc. For example, the *IsOperational* state holds

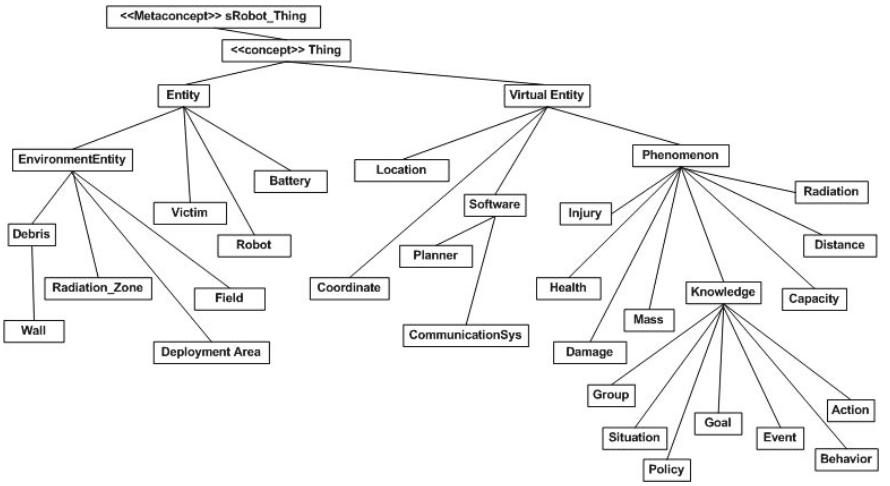

Fig. 2. Swarm Robotics Ontology Specified with KnowLang

when the robot's battery (the *rBattery* property) is not in the *batteryLow* state and the robot itself is not in the *IsDamaged* state.

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

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

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

The following is the specification of a policy called *ProtectVictimAgainst Radiation*. As shown, the policy is specified to handle the *Protect_Victim_ against_Radiation* goal and is triggered by the situation *VictimNeedsHelp*.

Further, the policy triggers via its $MAPPING$ sections conditionally the execution of a sequence of actions. When the conditions are the same, we specify a probability distribution among the $MAPPING$ sections involving same conditions (e.g., $PROBABILITY\{0.6\}$), which represents our initial belief in action choice.

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

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

5 Conclusion and Future Work

Swarm robotics systems generally exhibit a number of autonomic features resulting in complex behavior and complex interactions with the operational environment, often leading to a need for self-adaptation. The need of self-adaptation arises when a system needs to cope with changes in order to ensure realization of its objectives. To properly develop such systems, it is very important to appropriately handle their self-adaptive behavior. In this paper, we have presented an approach to capturing the requirements for, and modeling self-adaptive behavior, of swarm robotics. We consider that self-adaptive behavior extends the regular goals of a system upstream with special self-* objectives in the form of system's

ability to automatically discover, diagnose, and cope with various problems. To formalize self-* objectives, the approach relies on the KnowLang language, a formal language dedicated to knowledge representation for self-adaptive systems.

Future work is mainly concerned with further development of the Autonomy Requirements Engineering approach along with full implementation of KnowLang, involving tools and a test bed for autonomy requirements verification and validation.

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 Lero–the Irish Software Engineering Research Centre.

References

1. ASCENS: ASCENS - Autonomic Service-Component Ensembles. ascens-ist.eu (2014). http://www.ascens-ist.eu/
2. Bonani, M., Baaboura, T., Retornaz, P., Vaussard, F., Magnenat, S., Burnier, D., Longchamp, V., Mondada, F.: marXbot, Laborotoire de Systemes Robotiques (LSRO), Ecole Polytechnique Federale de Lausanne. mobots.epfl.ch (2011). http://mobots.epfl.ch/marxbot.html
3. Serbedzija, N., Hoch, N., Pinciroli, C., Kit, M., Bures, T., Monreale, G., Montanari, U., Mayer, P., Velasco, J.: D7.3: Third Report on WP7 Integration and Simulation Report for the ASCENS Case Studies. ASCENS Deliverable (2013)
4. Vassev, E., Hinchey, M.: Autonomy requirements engineering. In: Proceedings of the 14th IEEE International Conference on Information Reuse and Integration (IRI 2013), pp. 175–184. IEEE Computer Society (2013)
5. Vassev, E., Hinchey, M.: Autonomy requirements engineering: A case study on the bepicolombo mission. In: Proceedings of C* Conference on Computer Science & Software Engineering (C3S2E 2013), pp. 31–41. ACM (2013)
6. Vassev, E., Hinchey, M.: On the autonomy requirements for space missions. In: Proceedings of the 16th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing Workshops (ISCORCW 2013). IEEE Computer Society (2013)
7. 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)
8. 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. ASCENS Deliverable (2012)