ascens
www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

JD2.1: Languages and Knowledge Models for Self-Awareness and Self-Expression

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 2.2 (30.7.2011)**

Lead contractor for deliverable: **IMT**
Author(s): **Rocco De Nicola (IMT) - editor, Matthias Hölzl (LMU), Michele Loreti (UNIFI), Alberto Lluch Lafuente (IMT), Ugo Montanari (UNIPI), Emil Vassev (UL) and Franco Zambonelli (UNIMORE)**

Reporting Period: **2**
Period covered: **October 1, 2011 to September 30, 2012**
Submission date: **November 12, 2012**
Revision: **Final**
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

Executive Summary

Self-adaptivity is considered a fundamental feature of autonomic systems. It is thus of fundamental importance to equip the different linguistic and semantic tools that are introduced for achieving the goal of the ASCENS project with the right abstractions and constructs for enabling components of ensembles to adapt to changing environments or to mutated goals. Obviously, adaptivity is possible only if the individual components have the appropriate knowledge (self-awareness) and the appropriate capability (self-expression). These features have to be made available at different stages during the development of autonomic components and at run-time. In particular they are relevant both when providing an abstract modeling of the autonomic system and when providing its concrete implementation. But these features do play a key rôle also when the knowledge handler is designed. This deliverable describes how such mechanisms are realized in ASCENS. We focus, in particular, in abstract modeling techniques (SOTA and GEM), knowledge representation models (KnowLang and soft constraints) and specification languages (SCEL). In order to provide a more uniform and coherent presentation all the approaches focus on a common scenario, dealing with robot swarms operating in highly variable, unknown environments and having to solve their tasks in a collaborative manner.

Contents

1 Introduction **5**

1.1 Self-adaptation. 5

1.2 Self-awareness, self-adaptation and self-expression in robot swarms. 6

1.3 Structure of the Deliverable. 7

2 Abstract Modelling **8**

2.1 SOTA 8

2.1.1 The Rationale behind SOTA 8

2.1.2 The SOTA Space 9

2.1.3 Goals and Utilities 10

2.2 GEM and POEM 11

2.2.1 Trajectory Space, Situations, Fluents 11

2.2.2 Goals, Utilities and Strategies 13

3 Knowledge Representation Modelling **15**

3.1 KnowLang 15

3.1.1 Structuring Knowledge with KnowLang 16

3.1.2 Modeling Self-adaptive Behavior with KnowLang 17

3.1.3 ASK and TELL Operators and KnowLang Reasoner 18

3.1.4 Case Study 19

3.2 Soft Constraints 20

3.2.1 Constraints and Adaptation 20

3.2.2 Hierarchical Constraints for KnowLang 22

3.2.3 Constraints and Emergent Knowledge 22

3.2.4 Soft constraints in the case study 22

4 Languages **23**

4.1 SCEL: Service Component Ensemble Language 24

4.2 Robotics scenario in SCEL 24

4.3 Running the scenario 27

4.4 Soft Constraints as a linguistic abstraction in SCEL 27

5 Conclusions **28**

1 Introduction

Self-adaptivity is considered a fundamental feature of autonomic systems. It is thus of fundamental importance to equip the different linguistic and semantic tools that are introduced for achieving the goal of the ASCENS project with the right abstractions and constructs for enabling components of ensembles to adapt to changing environments or to mutated goals. Obviously, adaptivity is possible only if the individual components have the appropriate knowledge (self-awareness) and the appropriate capability (self-expression). These features have to be made available at run time and at the different stages during the development of autonomic components: In particular they are important both when providing an abstract modeling of the autonomic system and when providing its concrete implementation. But these features do play a key rôle also when the knowledge handler is designed.

This deliverable describes how such mechanisms are realized in ASCENS. We focus, in particular, in abstract modeling techniques (SOTA and GEM), knowledge representation models (KnowLang and soft constraints) and specification languages (SCEL).

1.1 Self-adaptation.

Self-adaptive systems have been widely studied in several disciplines ranging from Biology to Economy and Sociology. They have become a hot topic in Computer Science in the last decade as a convenient solution to the problem of mastering the complexity of modern software systems, networks and architectures. In particular, self-adaptivity is considered a fundamental feature of *autonomic systems*, that can specialize to several other self-* properties like self-configuration, self-optimization, self-protection and self-healing, as discussed for example in [20]. The literature on this topic enjoys valuable works aimed at capturing the essentials of adaptivity both in the most general sense (see e.g. [23]) and in specific fields such as that of software systems (see e.g. [39, 21]) providing in some cases very rich surveys and taxonomies.

Despite of all these efforts, there is no general agreement on the notion of adaptivity in general or in software systems, and no general consensus is perceived around a foundational model for adaptivity. According to widely accepted (though not very concrete as we shall see) definitions a software system is called "self-adaptive" if it *"modifies its own behavior in response to changes in its operating environment"* [34], where such "environment" or "context" has to be understood in the widest possible way, including both the external environment and the internal state of the system itself. Typically, such changes are applied when the software system realizes that *"it is not accomplishing what the software is intended to do, or better functionality or performance is possible"* [38].

The limited success obtained in the investigation of the foundations of (self-)adaptive software systems might be due to the fact that it is not clear what are the characterizing features that distinguish such systems from plain ("non-adaptive") ones. In fact, almost any software system can be considered self-adaptive, according to the definitions recalled above, since any system of a reasonable size can *modify its behaviour* (for example by following different branches at the same control point) as a *reaction to a change in its context of execution* (like the change of variables or parameters).

The ASCENS project proposes concrete notions of adaptation from two perspectives: the black-box one and the white box one. Early versions of both approaches and a brief discussion are described in [3] and summarized below.

Black-box Adaptation Some approaches adopt the above definitions from a black-box (or *behavioral* or *observational*) perspective (e.g. [19]), aimed at measuring how well a software system adapts to some context for some purpose.

Such perspective focuses on the point of view of an observer and does not care about the internal

mechanisms by which the adaptive behavior is achieved. On the one hand, this standing is interesting and useful for estimating or predicting the system robustness under some conditions. On the other hand it is of little use for design purposes where modularization and reuse are critical aspects.

White-box adaptation Instead, *white-box* perspectives allow one to inspect, to some extent, the internal structure of a system. They offer a clear *separation of concerns* to distinguish the cases where the changes of behaviour are part of the application logic from those where they realize the adaptation logic, calling adaptive only systems capable of the latter.

In general, the behavior of a component is governed by a program and according to the traditional, basic view, a program is made of *control* (i.e. algorithms) and *data*. Therefore, one can say that control and data are two conceptual ingredients that in presence of sufficient resources (like computing power, memory or sensors) determine the behaviour of the component. The conceptual notion of adaptivity of [7] requires to make explicit the fact that the behaviour of a component depends on some well identified *control data* which can be changed to *adapt* the component's behaviors. This definition of adaptation is then very simple but quite concrete: Given a component with a distinguished collection of control data, *adaptation* is the run-time modification of such control data.

From this basic definition one can immediately derive several others. A component is *adaptable* if it has a distinguished collection of control data that can be modified at run-time. Further, a component is *adaptive* if it is adaptable and its control data are modified at run-time, at least in some of its executions. Moreover, a component is *self-adaptive* if it modifies its own control data at run-time.

The intrinsic subjectivity of adaptation is captured by the fact that the collection of control data of a component can be defined, at least in principle, in an arbitrary way, ranging from the empty set ("the system is not adaptable") to the collection of all the data of the program ("any data modification is an adaptation"). This means that the white-box perspective is as subjective as black-box one. The fundamental difference lies in the responsible of declaring which behaviours are part of the adaptation logic and which not: the observer (black-box) or the designer (white-box).

Summarizing, in order to adhere to the proposed white-box notion of adaptivity, when modelling or specifying the behaviour of a system the designer can declare explicitly which behaviours are adaptations simply by designating as control data exactly those data whose modification triggers, according to his/her interpretation, an adaptation of the system.

1.2 Self-awareness, self-adaptation and self-expression in robot swarms.

In order to provide a more uniform and coherent presentation, all the approaches will focus on a common scenario where *self-adaptation*, *self-expression* and *self-awareness* appear as suitable mechanisms to achieve a successful adaptive behavior. These three mechanisms should not be considered as unrelated or independent. In fact, self-adaptivity and autonomicity mostly rely on an intelligent cooperation between declarative aspects of the system, that should be able to keep at run time a possibly abstract, up-to-date representation of the operating environment (context-) and of the system itself (self-awareness), and the procedural aspects, that should be able to transfer the relevant modification of the declarative representation into correspoding adaptation or reconfiguration actions (self-expression).

Robot swarms operating in highly variable, unknown environments and having to solve their tasks in a collaborative manner are not only one of the case studies (see [50, 49]) of the ASCENS project but also an archetypal example of collective self-adaptive systems. Consider, for example, the case of a robot that breaks down or gets trapped while performing its duties. Suppose further that the robot starts emitting a rescue request in form of some signal. The members of the swarm that detect the request have to decide whether to postpone their current activities and help the blocked robot.

Those robots that decide to start the rescue operation will need to face several problems, including the presence of obstacles in the terrain which may hamper navigation or disturb the rescue request signal. In order to be successful they have to adapt their behavior to such obstacles or other unexpected events (e.g. low battery, other help requests). That is, they cannot just run straightforwardly to the help request signal but may need to self-adapt their routes and behaviors independently (e.g. by circumventing the obstacle) or collectively (e.g. by forming assemblies or convoys or observing the trajectories of other robots). The latter may require complex forms of collective adaptation, called *self-expression* [51, 52], involving the coordination of individual adaptation strategies (e.g. enforcing the change of one adaptation pattern to another, changing the shape of an assembly or the leader of a convoy, etc.). All the decisions involved in this situation will be based on the *awareness* of the robot, i.e. on its knowledge of the environment (*context-awareness*) and "about its own entities, current states, capacity and capabilities, physical connections and relations with other systems in its environment" (*self-awareness*) [47, 16].

The scenario is inspired by the experiments presented in [33], which were conducted on the SWARM-BOT robotic platform [26] and is one of the three main scenarios considered in ASCENS . We assume that the current situation is that of an irregular ground area where some robots have been deployed, one of which is trapped in a small hole. We further assume that the trapped robot is able to emit a rescue-request light. At least one of the remaining robots is able to sense that signal, while the rest cannot. Unfortunately, the rescue signal consumes much more battery than ordinary signaling due to the required intensity. The terrain between the safe robots and the trapped one is full of obstacles: rocks of various size that have to be circumvented or climbed and might hide the light signal, and holes, again of various size. The number of robots needed to pull the trapped robots outside the hole is unknown. Indeed most of the above described information is unknown to the robots, but their knowledge base can of course contain useful information to provide estimates (e.g. to estimate the probability of successfully passing over a hole or the weight of a rock, etc.).

1.3 Structure of the Deliverable.

Section 2.1 is devoted to SOTA, a formalism for describing self-adaptation and self-awareness requirements. Self-adaptation is approached from a black-box perspective, by expressing the desired behavior of a system in terms of the *State Of The Affairs*, i.e sets of trajectories in the space of possible system configurations. A configuration represents all the relevant information of a system, including the internal status of components and their environment. In this manner SOTA specifications specify both the relevant information the system should be aware of (i.e. its self-awareness), and which trajectories the adaptive behavior of the system should guarantee (i.e. its self-adaptation requirements).

Section 2.2 overviews the GEM model, which provides a more detailed and formal counterpart of SOTA specifications. The treatment of self-adaptation and self-awareness is also based on identifying the relevant information the system should be aware of and considering trajectories in the corresponding space. In addition, the GEM approach allows to provide more detailed specifications of the system's adaptive behavior by means of specifications in an *action planning* language called POEM, where one can specify the system domain (i.e. the signature for the SOTA/GEM space) and the behavior of components, of ensembles and of the environment (by means of action rules).

Section 3.1 introduces the KnowLang framework, which supports the development of rich, ontology-based knowledge structures as well as the implementation of reasoning mechanisms. The latter are enacted by using a set of so-called policies, implementing a form of reactive reasoning. Policies are triggered by situations and enable the execution of actions. In this framework, adaptation is realized mainly in two ways: (1) via Tell actions, that update the knowledge-base, and (2) via effect-driven re-computation of actions' preference/probability, implementing a form of reinforcement learning.

Section 3.2 advocates soft constraints as a suitable mechanism for knowledge representation and reasoning. Soft programming addresses the problem of knowledge representation, reasoning, and adaptation in a uniform way: by providing a theory of constraints and its resolution, based on a mechanism known as constraint propagation. Constraints can be defined on several domains and are satisfiable up to a degree (soft), thus allowing for preferential/prioritized reasoning. Constraint propagation is a well-understood and modular mechanism that is well suited for domains that require to distribute both the knowledge representation and the reasoning tasks.

Section 4 provides a brief description of the specification/programming language SCEL, a *Service Component Ensemble Language*, and shows how it can be used to implement the case study described above by providing the SCEL code for significative parts of the code of the different robots. Two kinds of programs are provided. Some are just simple SCEL specifications that thanks to the formal operational semantics of the languages are amenable to many qualitative and quantitative analysis. Others are Java programs that exploit the language extension designed for dealing with key abstractions for autonomic computing such as group based communication, knowledge manipulation, etc.. The direct correspondence between the SCEL and the Java statements permits gaining confidence on the quality of the executable code, once key properties have been assessed at the specification level.

2 Abstract Modelling

2.1 SOTA

A key open issue in the study of self-adaptive systems concerns the identification of a general requirements modeling framework upon which to ground software analysis and development activities. To tackle this issue, we defined SOTA ("State Of The Affairs") [1], a robust conceptual framework that can act as an effective support to self-adaptive software development.

2.1.1 The Rationale behind SOTA

Traditionally, in the software engineering area, the requirements can be divided into two categories: functional requirements (what the system should do) and non-functional requirements (how the system should act in achieving its functional requirements, e.g., in terms of system performances, quality of service, etc.).

In the area of adaptive systems, and more in general of open-ended systems immersed in dynamic environments both functional and non-functional requirements are better expressed in terms of "goals" [30]. A goal, in general terms, is a "state of the affairs" that an entity aims to achieve.

The idea of goal-oriented modeling of requirements naturally matches goal-oriented and intentional entities (e.g., humans, organizations, and multi-agent systems). Therefore, since self-adaptation is naturally perceivable as an "intentional" quality, goal-oriented modeling appears the natural choice for self-adaptive systems, since it matches the "observable" intentionality of a system that takes actions aimed at adapting its behavior.

Interestingly, goal-oriented modeling of self-adaptive systems enables a uniform and comprehensive way of modeling functional requirements and non-functional ones, the former representing the eventual state of affairs that the system has to achieve, and the latter representing the current state of the affairs that the system has to maintain while achieving the goal.

As for the "state of the affairs" (from which the SOTA acronym derives), this represents the state of everything in the world in which the system lives and executes that may affect its behavior and that is relevant w.r.t. its capabilities of achieving. We could also say that such state of affairs is the "context" of the systems.

Against this background, SOTA builds on the most assessed approaches to goal-oriented requirements engineering [30]. For modeling the adaptation dimension, SOTA integrates and extends recent approaches on multidimensional modeling of context such as the Hyperspace Analogue to Context (HAC) approach [37]. In particular, such generalization and extensions try to account for the general needs of dynamic self-adaptive systems and components.

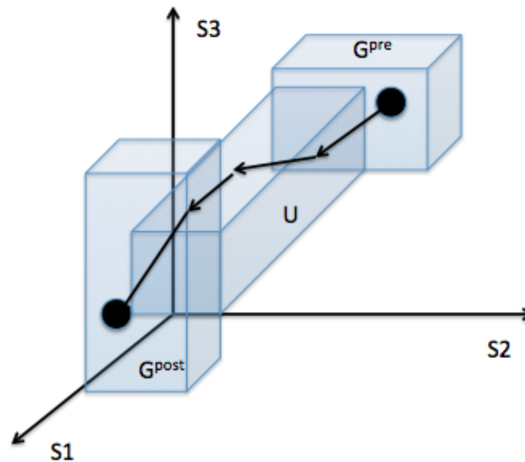

Figure 1: The trajectory of an entity in the SOTA space, starting from a goal precondition and trying to reach the postcondition while moving in the area specified by the utility.

2.1.2 The SOTA Space

SOTA assumes that, for an entity e (let it be an individual component or an ensemble), its current "state of the affairs" $S_e(t)$ at time t (or, for the sake of simplifying the notation, simply $S(t)$), can be described as a tuple of n values, each representing a specific aspect of the current situation:

$$S(t) = \langle s_1, s_2, \ldots, s_n \rangle$$

As the entity executes, S changes either due to the specific actions of e or because of the dynamics of e's environment. Thus, we can generally see this evolution of S as a movement in a virtual n-dimensional space $\mathbf{S}$ (see Fig.1):

$$\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times \ldots \times \mathbf{S}_n$$

Or, according to the standard terminology of dynamical systems modeling, we could consider $\mathbf{S}$ as the phase space of e and its evolution (whether caused by internal actions or by external contingencies) as a movement in such phase space.

To clarify, let us refer to the robotics case study and to the problem – described in the introduction – of robots in charge of reaching a target in an adaptive way. There, it is clear that a robot lives in a multidimensional SOTA space whose dimensions have to include the physical dimensions (i.e., the spatial coordinates of the environment). However, other dimensions that may have to be fruitfully included in the SOTA space may be the level of the batteries of a robot, its current speed, or the values of some parameters sensed by its sensors. In fact, all these parameters may affect the capability of a robot to properly execute and achieve what it was deployed for.

To model the evolution of the system in terms of “transitions”, $\theta(t, t+1)$ expresses a movement of e in the space $\mathbf{S}$, i.e.,

$$\theta(t, t+1) = \langle \delta s_1, \delta s_2, \ldots, \delta s_n \rangle, \delta s_1 = (s_1(t+1) - s_1(t))$$

A transition can be endogenous, i.e., induced by actions within the system itself, or exogenous, i.e., induced by external sources. The existence of exogenous transitions is particularly important to account for, in that the identification of such source of transitions (i.e., the identification of which dimensions of the SOTA space can induce such transition) enables identifying what can be the external factors requiring adaptation.

For instance, in the robotic case study, any movement of the robot induces a transition in the SOTA space, there included moving along the axis of the SOTA space representing the physical environment, but also moving down the axis representing the battery level.

2.1.3 Goals and Utilities

A *goal* by definition is the eventual achievement of a given state of the affairs. Therefore, in very general terms, a specific goal G_i for the entity e can be represented as a specific point, or more generally as a specific area, in such space. That is:

$$G_i = A_1 \times A_2 \times \ldots \times A_n, A_i \subseteq \mathbf{S}_i$$

In the case of a robot, a goal could be reaching a specific region of the physical space with a minimum level of batteries.

A goal G_i of an entity e may not necessarily be always active. Rather, it can be the case that a goal of an entity will only get activated when specific conditions occur. In these cases, it is useful to characterize a goal in terms of a precondition G_i^{pre} and a postcondition G_i^{pre}, to express when the goal has to be activated and what the achievement of the goal implies. Both G_i^{pre} and G_i^{post} represent two areas (or points) in the space $\mathbf{S}$. In simple terms: when an entity e finds itself in G_i^{pre} the goal gets activated and the entity should try to move in $\mathbf{S}$ so as to reach G_i^{post}, where the goal is to be considered achieved (see Fig.1). A goal with no precondition is like a goal whose precondition coincides with the whole space, and it is intended as a goal that is always active.

In the robotics case study, it could be the case that only when a robot “hears” via some sensor a specific help signal from another robot (pre-condition) it activates the goal of reaching that robot to rescue (post-condition).

As goals represent the eventual state of the affairs that a system or component has to achieve, they can be considered functional requirements. However, in many cases, a system should try to reach its goals by adhering to specific constraints on how such a goal can be reached. By referring again to the geometric interpretation of the execution of an entity as movements in the space $\mathbf{S}$, one can say that sometimes an entity should try (or be constrained) to reach a goal by having its trajectory confined within a specific area (see Fig.1). We call these sorts of constraints on the execution path that a system/entity should try to respect as *utilities*. This is to reflect a nature that is similar to that of non-functional requirements.

Like goals, a utility U_i can be typically expressed as a subspace in $\mathbf{S}$, and can be either a general one for a system/entity (the system/entity must always respect the utility during its execution) or one specifically associated with a specific goal G_i (the system/entity should respect the utility while trying to achieve the goal). For this latter case, the complete definition of a goal is thus:

$$G_i = \{G_i^{pre}, G_i^{post}, U_i\}$$

For instance, in the robot case study, a utility can express the need to move while respecting a minimal and a maximal speed. That it, by moving along a trajectory in the SOTA space that remains within a specific interval of the speed dimension in the SOTA space.

In some cases, it may also be helpful to express utilities as relations over the derivative of a dimension, to express not the area the trajectory should stay in but rather the *direction* to follow in the trajectory (e.g., try to minimize execution time, where execution time is one of the relevant dimension of the state of affairs). It is also worth mentioning that utilities can derive from specific system requirements or can derive from externally imposed constraint.

A complete definition of the requirements of a system-to-be thus implies identifying the dimensions of the SOTA space, defining the set of goals (with pre- and postcondition, and possibly associated goal-specific utilities) and the global utilities for such systems, that is, the sets:

$$\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times \ldots \times \mathbf{S}_n$$

$$\mathbf{G} = \{G_1, G_2, \ldots, G_m\}$$

$$\mathbf{U} = \{U_1, U_2, \ldots, U_p^e\}$$

Of course, during the identification of goals and utilities, it is already possible to associate goals and utilities locally to specific components of the system as well as globally, to the system as a whole. Thus, the above sets can be possibly further refined by partitioning them among local and global ones.

2.2 GEM and POEM

SOTA is concerned with the overall domain and the requirements of the system. For this it is sufficient to deal with the state of the affairs without regard for details such as the state's internal structure or the probabilities of the different trajectories. For a more detailed investigation of the structure and behavior of ensembles we need a more expressive model. To this end, in parallel with the definition of the SOTA model and in concert with it, we have defined the *General Ensemble Model (GEM)* [19], a mathematical model for the behavior of ensembles in the state-of-the-affairs space, and the *Pseudo-Operational Ensemble Modeling Language (*POEM*)*, a modeling language based on GEM (and thus SOTA). In Sect. 2.2.1 we introduce the notion of trajectory space on which the GEM model is based and describe its relationship to situations and time in POEM; in Sect. 2.2.2 we show how goals and utilities are used in GEM and POEM. In order to simplify the exposition we restrict ourselves to a deterministic scenario, even though both GEM and POEM permit probabilistic extensions.

2.2.1 Trajectory Space, Situations, Fluents

In general, a system can behave in a non-deterministic manner and therefore have multiple possible trajectories through the state space. If we know all possible trajectories of the system, we know everything that the state space can express about the system.[1] In GEM we identify a system $\mathbf{Sys}$ with the set of all its possible trajectories in the SOTA space which we call the system's *trajectory space* Ξ. Writing T for the time domain, $\mathbf{S}$ for the state space, and $\mathcal{F}[T \to \mathbf{S}]$ for the set of all functions from T to $\mathbf{S}$, the trajectory space is simply $\Xi = \mathcal{F}[T \to \mathbf{S}]$. Then, a system is a subset of the trajectory space, $\mathbf{Sys} \subseteq \Xi$.

The state of the affairs concept of SOTA can therefore also be expressed in an enriched way to account for such trajectories: for each trajectory ξ of the system, and at each point in time t the state

[1] This does not hold in the probabilistic case where the probability distribution of trajectories gives additional information.

of affairs is the value $\mathbf{Sys}(\xi, t)$, which is a point of the state space $\mathbf{S}$ which we assume to be a product of sets with the (finite or infinite) index set I:

$$\mathbf{Sys}(\xi, t) = \xi(t) = \langle s_i \rangle_{i \in I} \in \mathbf{S} \quad \text{if } \xi \in \mathbf{Sys}.$$

In GEM we structure the state space as the result of an interaction between the ensemble and its environment. We formalize this using the notion of a combination operator: let Ξ^{ens} and Ξ^{env} be the trajectory spaces of the ensemble and environment, respectively[2], and let $\otimes : \Xi^{ens} \times \Xi^{env} \to \Xi$ be a partial map that is a surjection onto $\mathbf{Sys}$, i.e., there exist $\mathbf{Sys}^{ens} \subseteq \Xi^{ens}$ and $\mathbf{Sys}^{env} \subseteq \Xi^{env}$ such that $\mathbf{Sys}^{ens} \otimes \mathbf{Sys}^{env} = \mathbf{Sys}$. In this case we obtain a trajectory of the system for compatible pairs of ensemble and environment trajectories in $\mathbf{Sys}^{ens} \times \mathbf{Sys}^{env}$. We therefore regard the system as the result of combining ensemble $\mathbf{Sys}^{ens}$ and environment $\mathbf{Sys}^{env}$ using the operator $\otimes$.

For example, in GEM we can structure a model of the robot ensemble as follows. First we define the state space $\mathbf{S}^{robot}$ as the Cartesian product of the robot's coordinates ($\mathbb{R}^2$) and its current state $\{\mathsf{Exploring}, \mathsf{Resting}, \mathsf{Trapped}\}$, and the trajectory space Ξ^{robot} as usual

$$\begin{aligned} \mathbf{S}^{robot} &= \mathbb{R}^2 \times \{\mathsf{Exploring}, \mathsf{Resting}, \mathsf{Trapped}\} \\ \Xi^{robot} &= \mathcal{F}[T \to \mathbf{S}^{robot}]. \end{aligned}$$

The model of each robot $\mathbf{Sys}_i^{robot}$ is a subset of the trajectory space consisting of all possible trajectories that the robot can take through its state space: $\mathbf{Sys}_i^{robot}$ thus belongs to the powerset of Ξ^{robot} (written $\mathfrak{P}(\Xi^{robot})$). The ensemble consisting of all N robots has as state space $\mathbf{S}^{ens} = (\mathbf{S}^{robot})^N$, as trajectory space $\Xi^{ens} = \mathcal{F}[T \to \mathbf{S}^{ens}]$, and the model of the ensemble, $\mathbf{Sys}^{ens}$, can be obtained from the models of the individual robots by a combination operator $\otimes : \mathfrak{P}(\Xi^{robot})^N \to \mathfrak{P}(\Xi^{ens})$ that combines all trajectories of robots that are physically possible, i.e., $\otimes$ is essentially the canonical map between $\mathfrak{P}(\Xi^{robot})^N$ and $\mathfrak{P}(\Xi^{ens})$, but it removes those trajectories where robots would overlap in space.

In this example, we define the state space for the environment as follows: We include the number of observed items of interest (e.g. obstacles) in the area $\mathsf{Items}^\sharp$, a function $\mathsf{Items}^{loc} : \mathbb{N} \to \mathbb{R}^2$ so that $\mathsf{Items}^{loc}(i)$ gives the location of the i-th item of interest that was discovered, and the coordinates of the robots to be rescued:

$$\mathbf{S}^{env} = \mathbb{N} \times \mathcal{F}[\mathbb{N} \to \mathbb{R}^2] \times \mathfrak{P}(\mathbb{R}^2).$$

As usual, the trajectory space of the environment is $\Xi^{env} = \mathcal{F}[T \to \mathbf{S}^{env}]$ and each environment $\mathbf{Sys}^{env}$ is a member of $\mathfrak{P}(\Xi^{env})$. In this simple example, the state space $\mathbf{S}$ for the whole ensemble is the product $\mathbf{S}^{ens} \times \mathbf{S}^{env}$ and the ensemble's trajectory space is defined as $\mathcal{F}[T \to \mathbf{S}]$; the combination operator for ensemble and environment has then the signature

$$\otimes : \mathfrak{P}(\Xi^{ens}) \times \mathfrak{P}(\Xi^{env}) \to \mathfrak{P}(\Xi)$$

and combines again all trajectories of the environment and the ensemble that are possible while removing those combined trajectories that cannot happen (e.g., no robot can be outside the explored area, and if no robot is in state $\mathsf{Exploring}$ during a time interval $[t_0, t_1]$, then the number of observed items cannot decrease between t_0 and t_1, etc.).

While the GEM model is conceptually simple, it quickly becomes tedious to write down the necessary relations and combination operators without additional linguistic constructs. We have therefore

[2] Formally we have $\Xi^{ens} = \mathcal{F}[T \to S^{ens}]$ where $S^{ens} = \prod_{k \in K} S_k^{ens}$, and $\Xi^{env} = \mathcal{F}[T \to S^{env}]$ where $S^{env} = \prod_{l \in L} S_l^{env}$. Note that the sets S_k^{ens} and S_l^{env} may be different from the sets S_i that appear in the system's state space $\mathbf{S} = \prod_{i \in I} S_i$.

defined the POEM language which simplifies the specification of a large class of GEM models in a style that resembles more traditional approaches to software development, without sacrificing the generality of GEM models.

A POEM model consists of two parts: a *domain specification* and behavioral *strategies*. The domain specification is a sorted first-order theory consisting of *background knowledge* that describes general knowledge about the domain, *constraints* on the domain and an *action theory* that describes which actions can be performed by components of the ensemble and the environment, and which effects they have on the domain. An interpretation of the domain specification gives rise to a GEM trajectory space; the possible execution traces of the model's strategies give rise to the GEM system. Therefore a POEM model formally corresponds to a class of GEM specifications.

POEM itself has few ontological commitments about the domain specification; the main restriction is that control of the system is effected by discrete *actions* that "change" the value of functional or relational *fluents*, and that the system is deterministic except for the choice of actions.[3] We call a sequence of actions a *situation*, and write $s = [a_1, \ldots, a_n]$ for the situation that consists of performing actions a_1 to a_n in this order and s_0 for the situation where no actions have been performed, $s_0 = []$. Each action a may have a *precondition* $poss(a, s)$ that specifies in which situations s the action a can be performed; often each action also has an associated *time*, written *time*(a) that specifies when the action takes place. Typically the domain of *time* is the GEM model's time domain T; in that case the situations describe the instances where direct control activities happen. We write $s :: a$ for performing action a in situation s, i.e., for performing a after performing all the actions in s. A fluent is a property of the environment that may change over time, which is formally represented as a function or relation with exactly one situation argument and no action argument; "changes" of the fluent happen if a fluent has different values for different situations.

Part of the domain specification for the robot domain is given in Fig. 2. To simplify the specification we ignore the time that a robot needs for moving. The functional fluent *location* describes the location of an object, *robot-state* describes a robot's state in a given situation; the relational fluent *found-item* is true if the robot has discovered an item. The action *move* is only possible when the robot is not currently trapped and if the goal location is not occupied by another robot.[4] Its effect is to take the robot to the specified position if there is no hole along the path; otherwise the robot becomes trapped in the location of the hole. Note that we do not specify domain closure axioms since these can be automatically generated. We expect that the domain specification will often be provided in the KnowLang language (see Sect. 3.1) which offers more elaborate constructs for knowledge representation, but can be translated into the POEM primitives in a straightforward manner. We discuss behavioral specifications after introducing goals and utilities in the next section.

2.2.2 Goals, Utilities and Strategies

GEM is intended to serve as semantic foundation for various kinds of calculi and formal methods which often have a particular associated logic. We define the notion of goal satisfaction "System $\mathbf{Sys}$ satisfies goal G," written $\mathbf{Sys} \models G$ in a manner that is parametric in the logic and in such a way that different kinds of logic can be used to describe various properties of a system (see [19] for details). In POEM models, goals correspond directly to relations in the domain specification, since the two possible interpretations for $\mathbf{Sys} \models G$ (as logical entailment and goal satisfaction in GEM) coincide.

Strategies in POEM serve to connect goal-based requirements specifications in the style of SOTA with concrete process models, such as those introduced by SCEL (see Sect. 4). POEM strategies con-

[3] Note that this restriction is not as severe as it may appear at first. In particular, it allows the specification of continuous deterministic control and probabilistic behaviors and control and therefore, e.g., of Markov Decision Processes with probabilistic (stationary and non-stationary) policies. It does not allow for continuous probabilistic control.

[4] We use the abbreviation $(\exists^{\neq} r_1, \ldots r_n).P$ for $(\exists r_1, \ldots r_n).r \neq r_1 \wedge \cdots \wedge r_{n-1} \neq r_n \wedge P$.

sort	*action*
sort	*situation*
sort	*position*
sort	*rstate*
sort	*obj*
subsort	*robot* $<$ *obj*
constant	r_i : *robot*
constant	p_i : *position*

fluent	*robot-state* : *robot* $\times$ *situation* $\to$ *rstate*	
fluent	*found-item* : *robot* $\times$ *situation*	
fluent	*ready?* : *robot* $\times$ *robot* $\times$ *situation*	
fluent	*location* : *obj* $\times$ *situation* $\to$ *position*	
action	$move(r, p)$: *robot* $\times$ *position* $\to$ *action*	
pre	$poss(move(r,p), s) \iff \big(\neg(robot\text{-}state(r,s) = \mathsf{Trapped})$	
	$\wedge\neg(\exists^{\neq} r' : robot).location(r', s) = p\big)$	
eff	$location(r, s :: move(r,p)) = q$	
	$\wedge\, robot\text{-}state(r, s :: move(r,p)) = \mathsf{Trapped}$	**if** hole at q
	$location(r, s :: move(r,p)) = p$	**otherwise**

Figure 2: POEM domain specification for the robot example

sist of preconditions, maintain goals and achieve goals, as well as an optional process description. Strategies, goals and processes are first class values that can be inspected, passed as parameters, dynamically created, loaded or communicated to other service components, etc. Strategies can refer to goals in the domain and, e.g., use them to select other strategies for achieving sub-goals; this enables a straightforward specification of goal-driven architectures as described in Sect. 2.1; the possibility to communicate goals and strategies between SCs together with the availability of a logical domain model enables sophisticated ways of achieving white-box adaptation. We call a strategy *fully specified* if it has a process description that either references no other strategy or only fully specified strategies.

POEM processes are specified in a non-deterministic, goal-directed, concurrent, higher-order action language. The family of action languages [40] is comprised of languages whose *primitive actions* correspond to actions in a logical domain specification; in order to execute a primitive action its precondition must be satisfied.[5] Programming and modeling-language constructs are layered on top of the primitive actions to allow the concise specification of behavior in the domain modeled by the domain specification. The POEM strategy language includes operators for non-deterministic choice of action or parameters and supports both *don't care (committed choice)* and *don't know* non-determinism; like most concurrent languages it defaults to committed choice. POEM supports concurrent execution of strategies, with a concurrency model based on that introduced by SCEL (see Sect. 4). If the primitive actions of the domain model and the reasoning process for computations with *don't know* non-determinism can be represented by operations on SCEL knowledge repositories, a fully specified POEM strategy corresponds to a SCEL process together with a specification of its preconditions and effects.[6]

To give a simple example we specify a strategy that generates a plan to satisfy the SOTA-Goal $G = \{G^{pre}, G^{post}, U\}$ whenever this is possible in the underlying action theory: The **pick-such-that-do** operation non-deterministically chooses a value for its argument that satisfies the condition given in its second clause and continues the execution of the strategy with a binding for the chosen value; **loop** is a non-deterministic loop that executes its body zero or more times; **holds?** checks a condition and fails if the condition is false. **wait** is similar to **holds?**, but blocks the strategy until its argument (a logical formula in the domain theory) becomes true. The operation **search** executes its body "off-line", using *don't know* non-determinism, therefore we obtain a plan to satisfy G by:

search(
  wait(G^{pre});
  loop(**pick**(a : *action*) **such-that holds?**(U) **do**(a));
  holds?(G^{post}))

[5] Note that this notion of action language is distinct from the action calculi introduced in [25].

[6] Since SCEL is parametric in the choice of model for knowledge repositories, the required reasoning about the domain can be captured in SCEL by defining appropriate semantics for the **get**, **put** and **qry** operations.

initially	$\mathit{location}(r_1, s_0) = p_1$
	$\big(\mathit{call\text{-}for\text{-}help}(r_1)^* \gg \mathit{help}(r_1)^* \gg \mathit{random\text{-}walk}(r_1)^*\big) \parallel \big(\mathit{call\text{-}for\text{-}help}(r_2)^* \gg \dots\big)$
strategy	$\mathit{random\text{-}walk}(r : \mathit{robot})$
proc	**while** $(\neg \mathit{found\text{-}item}(r, \mathbf{now}))$ **do** **pick**$(p : \mathit{position}); \mathit{move}(r, p)$ **end**
goal	$\mathit{found\text{-}item}(r, \mathbf{now})$
strategy	$\mathit{help}(r : \mathit{robot})$
pre	$(\exists^{\neq} r' : \mathit{robot}).\mathit{robot\text{-}state}(r', \mathbf{now}) = \mathsf{Trapped}$
bind	$r_t : \mathit{robot}$
proc	**pick**$(r_t : \mathit{robot})$
	such-that $\mathit{robot\text{-}state}(r_t, \mathbf{now}) = \mathsf{Trapped}$;
	pick$(p : \mathit{position})$
	such-that $(\mathit{adjacent}(p, \mathit{location}(r_t, \mathbf{now})) \wedge \neg(\exists^{\neq} r'' : \mathit{robot}).\mathit{location}(r'', \mathbf{now}) = p)$
	do $\mathit{move}(r, p)$;
	$\mathit{ready\text{-}to\text{-}rescue}(r, r_t)$;
	wait $((\exists^{\neq} r_1, r_2).\mathit{ready?}(r_1, r_t, \mathbf{now}) \wedge \mathit{ready?}(r_2, r_t, \mathbf{now}))$;
	$\mathit{rescue}(\{r, r_1, r_2\}, r_t)$
goal	$\neg \mathit{robot\text{-}state}(r_t, \mathbf{now}) = \mathsf{Trapped}$

Figure 3: POEM behavioral specification for the robot example

This purely planning-based approach is obviously unsuitable for most domains since the search space of all actions is prohibitively large even for modest problems. Therefore typical POEM strategies consist of a mix of pre-defined behaviors and non-deterministic choices. Some strategies for the robot scenario are shown in Fig. 3.

The strategy *random-walk* shows a possible implementation of a random walk until the robot has found an item. It uses non-deterministic choice to select the next position and then executes the primitive action *move* to move the robot to the chosen position. The **while**-loop ensures that this behavior is repeated until the robot has discovered an item. Note that the **goal** clause in the strategy specifies only partial correctness; there is no guarantee that the robot will ever discover an item, and the strategy also does not take into account the possibility of the robot becoming trapped.

Instead of integrating the behaviors for a breakdown and for providing help into a single strategy we prefer to model these concerns by two strategies executing concurrently with the basic *random-walk* strategy. Concurrent execution of strategies can be controlled by policies; for simplicity we use two operators in the example: $\parallel$ represents unrestricted interleaving and is used to model the behavior of independent robots; $S_1 \gg S_2$ is a concurrent form of execution where strategy S_1 is performed whenever possible, and S_2 is only executed when S_1 is finished and its precondition is not applicable. The strategy *call-for-help* has the highest priority, so no other strategy will be tried while the robot is trapped. The strategy *help* has priority over *random-walk*, so whenever a robot knows about another robot being trapped it will try to help rather than continue to explore. This strategy also shows how several robots can synchronize using the shared fluent *ready?*.

A more detailed discussion of POEM can be found in the Technical Report [18].

3 Knowledge Representation Modelling

3.1 KnowLang

Developing intelligent systems with Knowledge Representation and Reasoning (KR&R) has been an increasingly interesting topic for years. Examples are found in semantic mapping [14], improving planning and control aspects [31], and most notably HRI systems [22, 17]. 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.

One of the main scientific contributions that we expect to achieve with ASCENS is related to KR&R where within the WP3's mandate we are currently developing the KnowLang framework. 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 [32]. The language aims at efficient and comprehensive knowledge structuring and awareness based on logical and statistical reasoning. It helps us to tackle [44] 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) [44] and knowledge representation for autonomic behavior [46]. By applying the KnowLang's multi-tier specification model we build a Knowledge Base (KB) structured in three main tiers [44]: 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.

3.1.1 Structuring Knowledge with KnowLang

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 and is hierarchically linked to other concepts through *PARENTS* and *CHILDREN* relationships. In addition, for reasoning purposes every concept specified with KnowLang has an intrinsic *STATE* attribute that may be associated with a set of possible *state values* the concept instances may be in. The concept instances are considered as objects and are structured in object trees. The latter are 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 *relations*. Relations connect two concepts, two objects, or an object with a concept 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 4 shows a KnowLang specification sample demonstrating both the language syntax [41] and its visual counterpart - a concept map based on interrelations with no probability distributions.

Modeling knowledge with KnowLang [45, 43] goes over a few phases:

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

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

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

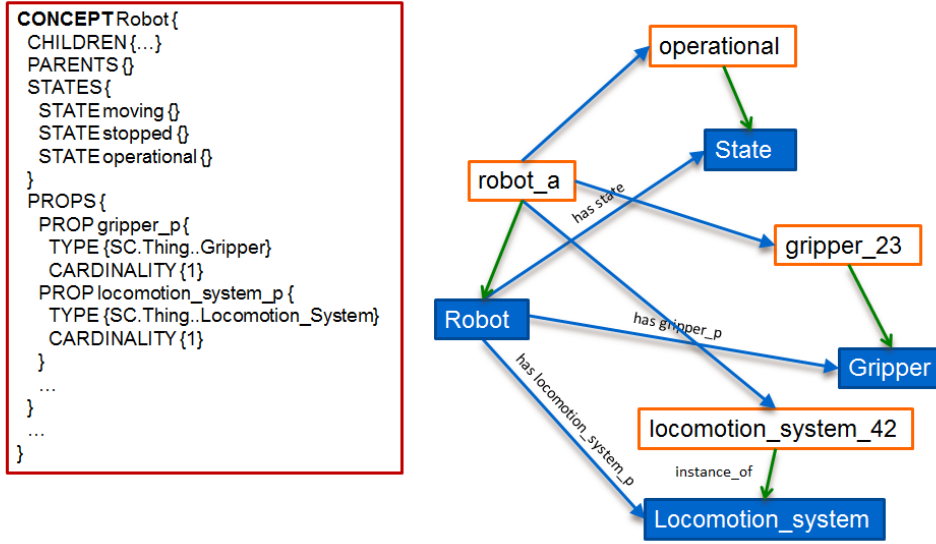

Figure 4: KnowLang Specification Sample

3.1.2 Modeling Self-adaptive Behavior with KnowLang

KnowLang employs special knowledge structures and a reasoning mechanism for modeling autonomic self-adaptive behavior [46]. Such a behavior can be expressed via KnowLang *policies*, *events*, *actions*, *situations* and *relations* between policies and situations (see Definitions 1 through 9). Policies (Π) are at the core of autonomic behavior. 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 2). A condition is a Boolean expression over ontology (see Definition 4), e.g., the occurrence of a certain event.

Policy situations Si_π are situations (see Definition 6) 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 2). Therefore, the optional policy-situation relations (R_π) justify the relationships between a policy and the associated situations (see Definition 9). Note that in order to allow for self-adaptive behavior, *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 to 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 $si \overset{[Z]}{\to} \pi$ - see Definition 9). Hence, the presence of *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 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 7).

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

Def. 2 $\pi :=< g, Si_\pi, [R_\pi], N_\pi, A_\pi, map(N_\pi, A_\pi, [Z]) >$ *(Policy)*
$A_\pi \subset A, N_\pi \overset{[Z]}{\rightarrow} A_\pi$ *(A_π - Policy Actions)*
$Si_\pi \subset Si, Si_\pi \overset{[R_\pi]}{\rightarrow} \pi \rightarrow N_\pi$ *(Si_π - Policy Situations)*
$R_\pi \subset R$ *(R_π-Policy-Situation Relations)*

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

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

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

Def. 6 $Si := \{si_1, si_2, ..., si_n\}, n \geq 0$ *(Situations)*

Def. 7 $si :=< s, A \overset{\leftarrow}{_{si}}, [E \overset{\leftarrow}{_{si}}], A_{si} >$ *(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)*

Def. 8 $R := \{r_1, r_2, ..., r_n\}, n \geq 0$ *(Relations)*

Def. 9 $r :=< \pi, [rn], [Z], si >$ *(Relation, rn - Relation Name, Z - Probability Distribution)*
$si \in Si, \pi \in \Pi, si \overset{[Z]}{\rightarrow} \pi$

KnowLang policies can be built to impose behavior based on POEM's strategies. 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 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*.

Please refer to D3.2 [48] for more details on the KnowLang specification formalism.

3.1.3 ASK and TELL Operators and KnowLang Reasoner

KnowLang is to be supplied with a special KnowLang Reasoner, which in addition to the trivial knowledge-querying and knowledge-updating tasks will support self-adaptive behavior retrieval. The KnowLang Reasoner operates in the KR Context (it operates with KR symbols only) and the system talks to the reasoner via the *ASK* and *TELL Operators* [42]. TELL Operators feed the KR context with important information driven by errors, executed actions, new sensory data, etc., thus helping the KnowLang Reasoner to update the KB with recent changes in both the system and execution environment. The system uses ASK Operators to receive recommended behavior (e.g., ASK_BEHAVIOR Operator [42]) 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.

For example, when called the ASK_BEHAVIOR Operator [42] will ask the Reasoner to generate a self-adaptive behavior by considering the actual situation the system is currently in. Thus, it looks up for the current situation by estimating the current system state and then evaluates the relations of that situation with policies to determine which policy to apply. There are also other variants of the ASK_BEHAVIOR Operator, e.g., the system may ask for a self-adaptive behavior to achieve a particular goal or a behavior that will lead the system out of a particular situation. Note that the ASK_BEHAVIOR operates exclusively in the KR Context and thus, the relevance of its output highly depends on the relevance of the knowledge stored in the KB. Thus, it is very important that the system feeds the KB with any important relevant information about the system itself and the execution environment, e.g., errors, sensory data, raised events, executed actions, etc. Therefore, if we assume that the system is implemented in SCEL, then in the program there should be explicit SCEL calls of both TELL and ASK Operators any time when the system has to pass/get information to/from the KB.

3.1.4 Case Study

To illustrate *autonomic self-adaptive behavior* based on this approach, we are going to elaborate on the "trapped robot case study" by assuming that the *trapped robot* keeps sending a help signal and Robot_A is receiving that signal. Eventually, the *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 π_1 (see Figure 5). Although we are missing the basic specification of the involved *actions*, *goal*, *situation* and *relation*, we can conclude that the current situation si_1:*"a robot needs assistance"* will trigger a policy π_1:*"go to the signal source"* if the relation $r_1(si_1, \pi_1)$ has the higher probabilistic belief rate. The π_1 policy will realize actions $Turn$ and $Move$ iff the robot's battery is charged at least 50% and there is no another 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., $\{Action.Turn(Action.GetSignalAngle), Action.Move\}$.

```
CONCEPT_POLICY π1 { // go to signal source
   CHILDREN {}  PARENTS {SC.Thing..Policy}
    SPEC {
      POLICY_GOAL { Goal.g1 } //reach signal source
      POLICY_SITUATIONS { Situation.si1 } //a robot needs assistance
      POLICY_RELATIONS { Relation.r1 } //relates π1 and s1
      POLICY_ACTIONS { Action.Turn, Action.Move }
      POLICY_MAPPINGS {
         MAPPING {
            CONDITIONS { Robot_A.Battery.level >= 0.5 AND
Action.GetPriorityTasks(Robot_A) = 0 }
            DO_ACTIONS {
               Action.Turn(Action.GetSignalAngle), Action.Move }
            PROBABILITY { 1 }
         }
} } }
```

```
CONCEPT_POLICY π2 { // avoid obstacle
   CHILDREN {}  PARENTS {SC.Thing..Policy}
    SPEC {
      POLICY_GOAL { Goal.g2 } //free road
      POLICY_SITUATIONS { Situation.si2 } //road is blocked
      POLICY_RELATIONS { Relation.r2 } //relates π2 and si2
      POLICY_ACTIONS {
         Action.TurnRight, Action.TurnLeft, Action.Move}
      POLICY_MAPPINGS {
         MAPPING {
            DO_ACTIONS { Action.TurnRight, Action.Move }
            PROBABILITY { 0.6 } }
         MAPPING {
            DO_ACTIONS { Action.TurnLeft, Action.Move }
            PROBABILITY { 0.4 }}
} } }
```

Figure 5: KnowLang Policies

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 π_2:*"avoid obstacle"* (see Figure 5). Policy π_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

π_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 π_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 π_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 π_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 π_2. Following π_2, the robot can fall again into si_3 and then back to si_2. However, every time when policy π_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 π_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.

3.2 Soft Constraints

This section describes our efforts to integrate constraint-based features to support or enrich knowledge and adaptation mechanisms. We start in Section 3.2.1 with a short overview of soft constraints and their use for adaptation purposes. Next (Section 3.2.2), we move to the integration of constraints within KnowLang [29], aimed at obtaining a more flexible way of specifying knowledge. Last (Section 3.2.3) we focus on need to deal with emergent and distributed knowledge, which requires the uses of distributed constraint handling mechanisms.

3.2.1 Constraints and Adaptation

Constraint Satisfaction Problems *A connection graph* [28] is a tuple $\langle N, A, a, c\rangle$, where N is a set of nodes, A is a set of arcs, $a \in A$ is an *interface* arc, and c is a connection function. *Connection function* $c = \bigcup_k (A_k \to N^k)$ is a correspondence between arcs and nodes: $c(h) = \langle x_1, \dots, x_k\rangle$ is a tuple of nodes connected by h, $x_i \neq x_j$ when $i \neq j$. Additionally, $A = \cup_k A_k$ is a ranked set: each $h \in A_k$ is an arc connected to k nodes. We refer to the connection graph by writing: $a \leftarrow G$, where $G = \langle N, A, c\rangle$. *Labeled* arcs are sometimes used to introduce additional functions in the tuple assigning a label to every arc. *A network of constraints* [28] is a pair $C = a \leftarrow G \mid l$, where $a \leftarrow G$ is a finite connection graph, whose nodes are variables and arcs are constraints; l is a labelling function $l : \bigcup_k (A_k \to P(U^k))$, where U is a finite set of values for the variables of C; $P(U^k)$ is the set of all k-relations on U. Solving a network of constraints is called a *constraint satisfaction problem (CSP)*. Let $C = a \leftarrow G \mid l$, $G = \langle N, A, a, c\rangle$, $n = \#N$, and $\langle x_1, \dots, x_n\rangle$ be any ordering of the variables of N. Given that $v = \langle v_1, \dots, v_n\rangle$ is n-tuple of values of U, let us set $v|_{\langle x_{i1}, \dots, x_{im}\rangle} = \langle v_{i1}, \dots, v_{im}\rangle$. The solution of network C is the set $\{\langle v_1, \dots, v_n\rangle|_{c(a)} : \forall b \in A, \langle v_1, \dots, v_n\rangle|_{c(b)} \in l(b)\}$. In other words, the CSP for a network of constraints is to find the set of all the assignments of the variables connected by the interface arc such that every such assignment can be extended to an assignment of all the variables in N which satisfies all the constraints in A [28].

Solving Constraint Satisfaction Problems Constraint propagation turns a constraint satisfaction problem into an equivalent one that is easier to solve [2] by enforcing some kind of local consistency. This can be done by applying a set of relaxation rules. Let $C = a \leftarrow G \mid l$ be a network of constraints. A *relaxation rule* r is any subgraph $r = b \leftarrow F$ of $a \leftarrow G$. Applying the relaxation rule r to C means solving the network $b \leftarrow F \mid l$ and then setting the labelling function of the corresponding subgraph as implied by the obtained solution. It was proved that a relaxation rule returns an equivalent network of constraints [28]. A generic *relaxation algorithm* works by applying a number of relaxation rules until no more changes can be done (in this case, we reach a *stable* network).

Given two disjoint graphs $a \leftarrow G$ and $a' \leftarrow G'$ with $G = \langle N, A, c\rangle$, $G' = \langle N', A', c'\rangle$, and an arc $b \in A$ with $rank(b) = rank(a')$, the *replacement* of b with $a' \leftarrow G'$ in $a \leftarrow G$ is the new graph $a \leftarrow H = a \leftarrow G[a' \leftarrow G']$ obtained by identifying b with a' and the tuple of nodes connected to b with the tuple of nodes connected to a'. A Hypergraph Replacement System (HRS) is a pair $\langle a \leftarrow G, P\rangle$ where $a \leftarrow G$ is the *initial* graph and P is a set of *productions* graphs. The *graph language* generated by a HRS is the set of graphs of the form $a \leftarrow H = a \leftarrow G[a_1 \leftarrow G_1] \ldots [a_n \leftarrow G_n]$, where graphs $a_i \leftarrow G_i, i = 1, \ldots, n$, are in P. Sequence $\langle a_1 \leftarrow G_1 , \ \ldots \ , \ a_n \leftarrow G_n\rangle$ is a *derivation* of $a \leftarrow H$.

It is possible to prove [28] that the relaxation algorithm which, beginning from a network of constraints $a \leftarrow H \mid l$, applies to it in the reverse order the relaxation rules corresponding to a derivation of $a \leftarrow H$, including the initial graph itself, namely the relaxation rules $a_n \leftarrow G_n$, ..., $a_1 \leftarrow G_1$ and $a \leftarrow G$, is *perfect*. A perfect relaxation algorithm applies every relaxation rule only once and the relation in the interface arc of the resulting graph is the solution of the initial network of constraints $a \leftarrow H \mid l$. As a consequence, in the finite case (all graphs, N and P being finite) a perfect relaxation algorithm provides a linear solution algorithm for any class of networks whose graphs are included in the language of some HRS. A perfect relaxation algorithm can be understood as an application of the dynamic programming solution method. Given a HRS $\mathcal{H}$, a *hierarchical* network of constraints is a network of constraints and a derivation sequence for its graph in $\mathcal{H}$.

From crisp to soft constraints and how to program them for adaption purposes Classical CSPs are not well-suited in several real-life scenarios. Indeed, CSPs are not able to model constraints that are preferences rather than strict requirements or to provide a ”non-complete” solution when the problem is over- constrained. A *soft* CSP [4] handles an enriched network of constraints where constrains rather than returning booleans yield more informative values, such as preference values, fuzzy values, probabilities or costs, which form a *constraint semiring*. Relaxation algorithms can be applied only in some cases, while dynamic programming (perfect relaxation) is always effective. Within ASCENS, soft constraints have been shown to be expressive enough to offer a declarative view of several optimization problems relevant to the e-mobility case study [27].

Ordinary logic programming (LP) can be considered as an extension of CSP where: (i) disjunction is available; (ii) the network structure is recursively defined; and (iii) predicates concern assignments to Herbrand terms rather than only to constants. LP can be extended to soft constraint LP [5] (SCLP), by extending predicates to functions yielding constraint semiring values. Soft *concurrent* constraint programming [6, 10] is a programming paradigm which includes, besides constraints, a procedural part, in process algebra style, and guard primitives like ask, tell and guarantee negotiation constructs. Future work is planned to integrate soft constraint programming into SCEL.

As explained in Section 1.2 adaptation often relies on a convenient interaction between procedural and declarative information. (Soft) constraint programming is a well studied approach where both aspects are structured and consistently interacting. Thus its relevance to adaptation is clear. In the *white box* approach to adaptation studied in ASCENS, at least some of the constraints should be considered as part of the *control* data, i.e. of those data that should be modified in the process of adaptation. If constraints are modeling nontrivial knowledge, constraint management could be complex, and certainly not monotone. However, as explained in Section 3.2.3, also the process of constraint propagation can be considered as an important mechanism for adapting and making consistent the behavior of systems.

3.2.2 Hierarchical Constraints for KnowLang

For the purpose of obtaining a more flexible way of specifying knowledge representation in ASCENS, an integration of the constraints paradigm with KnowLang has been recently proposed [29]. The approach extends KnowLang with a technique where knowledge can be enriched as special restrictive rules that may require full or partial satisfaction, and represent special liveness properties.

The approach has been applied to derive a knowledge representation structure for the marXbot mobile robotics platform [29]. The hierarchical structure of the ontology is reflected in the tree structure (where additionally the branches are connected through a bounded number of nodes) of the (soft) constraint network.

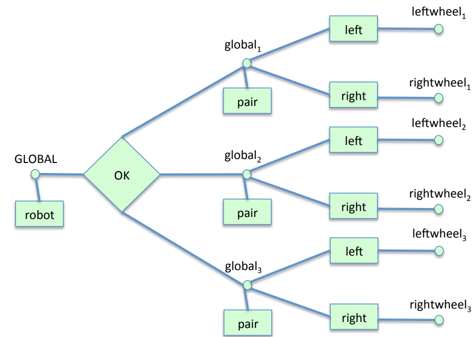

Figure 6: A network of constraints.

In general, the comprehensive structuring of KnowLang should be matched by a corresponding articulation of the soft CSP part. Hierarchical networks of constraints are well suited for this purpose since, as we mentioned, they are equipped with linear solution algorithms. For instance, the specification of the locomotion system of the marXbot robot [29] includes the following two clauses.

```
robot(GLOBAL) :- OK(GLOBAL,global1,global2,global3),
                   pair(global1),pair(global2),pair(global3)
pair(global) :- left(global,leftwheel),right(global,rightwheel)
```

They can actually be interpreted as productions, where *pair* and *robot* arcs are the interface arcs. The generated network is shown in Figure 6. Notice that the second production has been employed three times, and that fresh nodes have been introduced. Notice that the graph is tree-like. This is due to the fact that all the interface arcs have rank 1. In general the structure of the generated graph is that of a *thick tree*, i.e. trees where the branches are connected through a bounded number of nodes.

3.2.3 Constraints and Emergent Knowledge

Very often knowledge is *emergent* and *distributed*: it changes over time; local knowledge does not correspond to global knowledge, and new information is unequally distributed. Additionally, knowledge may be uncertain or not completely available.

Each service component can represent knowledge as a set of constraints about himself and the surrounding world: (K_1, C_1), (K_2, C_2), $\ldots$, where C_i are the constraints, and K_i are the keys that explain the meaning of the constraints (e.g. distance, temperature, danger level). Constraints are a generic way to represent knowledge, and constraint operations correspond to knowledge operations.

As explained in [36] knowledge representation styles can be integrated, and distributed constraint handling techniques can carry on deduction steps and consistency checks, where the local, specific knowledge interacts with the constraint representation via suitable interfaces.

Emergent knowledge needs to be discovered and combined with existing knowledge. When a service component discovers new information, its local knowledge can propagate to become global. The process starts with neighbours by achieving some kind of local consistency, and propagates further.

3.2.4 Soft constraints in the case study

Consider the case study of this deliverable and the situation depicted in Figure 7 (a), where where robots (small circles) must avoid the obstacles A and B. Each robot keeps track of the distance and the angle to the obstacle. Because of measure errors, a robot uses an estimation of its own location

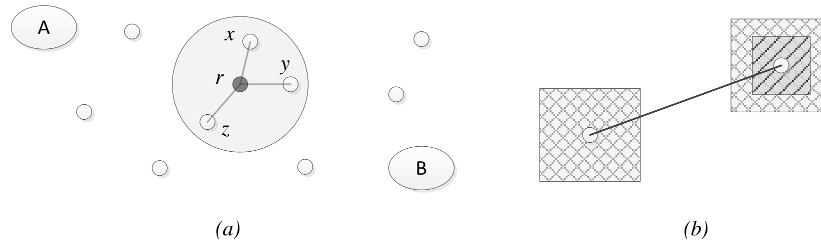

Figure 7: *(a)* Robot r and all robots within the range (x, y, z) form an ensemble, and adapt their control data concurrently. *(b)* Two robots combine their control data represented as constraints.

with a specific degree of certainty. With the distance, uncertainty grows. We consider a 10% error: a measure error up to 1 cm for each 10 cm.

In order to estimate its own position over time, robots may apply *social odometry* [15] techniques, where robots exchange location information with robots within communicating range (i.e. *ensembles*), and adjust their own estimation accordingly. While changes in data about orientation and distance when moving according to robot plans can be considered as ordinary evolution, changes to their planning data as a consequence of interaction with other robots should be seen as adaptation.

We use a rectangle to represent a robot's beliefs about his own location: a robot believes he can be located anywhere within the rectangle. We can see this as a constraint. With a higher uncertainty, the rectangle will be larger. Our assumption is that the distance to an obstacle is uncertain and needs to be adjusted. When two robots meet, they combine their location information in the following way (Figure 7b): knowing the distance and the angle between the two robots, we move one rectangle in the direction to the other, and compute an intersection. A smaller rectangle can be received as a result. We can see this as achieving some kind of local consistency, or propagation of constraints. Constraint propagation can be computed concurrently for all robots within the ensemble.

This example highlights the convenience of constraint propagation in scenarios with distributed and emergent knowledge. To deal with more sophisticated scenarios, we may proceed in many ways, both in terms of the knowledge actually represented in the constraints, and in terms of the kind of global effects which can emerge from propagation. The linguistic constructs of SCEL look quite convenient, with the logically defined notion of ensemble, for modeling both the dynamic interaction taking place during constraint propagation and the structural nature of hierarchical constraints.

4 Languages

Models of computation, and languages, for self-aware, self-adaptive and self-expressive autonomic components and ensembles need to include: (i) procedural components; (ii) declarative knowledge representation components and their bookkeeping primitives; and (iii) primitives to allow for the interaction of the two components. The latter part requires innovative ideas, since adaptivity and autonomicity rely mostly on an intelligent cooperation between procedural and declarative aspects of system behavior. To this aim, in this section we briefly sketch a dialect of the SCEL language, that relies on a simple notion of knowledge structured as a set of data tuples. There is obviously a continuum of alternatives between this basic choice that leaves all control to the programming language and the possibility of delegating all decisions to the knowledge handler modelled in the style of KnowLang. At the end of this section, we perform an initial step in the direction of having a "more active" knowledge handler by briefly considering the impact of an approach that relies on constraint-based knowledge. For a more detailed account of the work on SCEL, the reader is referred to [35].

4.1 SCEL: Service Component Ensemble Language

SCEL [11, 13, 12] provides abstractions explicitly supporting autonomic computing systems in terms of *Behaviors*, *Knowledge* and *Aggregations*, according to specific *Policies*. *Behaviors* describe how computations progress. *Knowledge* provides the high level primitives to manage pieces of relevant information coming from different sources, a distinction is assumed between *application data* and *control data*, the latter being used exactly for guaranteeing self-awareness and adaptation. *Aggregations* describe how different entities are brought together to form *components* and *ensembles* and to offer the possibility to construct the *software architecture* of autonomic systems. *Policies* control and adapt the actions of the different components. For the sake of simplicity, in this section, we do not consider policies explicitly while assuming that all the interactions are always permitted.

In SCEL, *behaviours* are modeled as *processes* executing actions, in the style of standard process calculi. Actions of the form $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ are used to withdraw, retrieve, add information items from/to the knowledge repository identified by c. In this deliverable we assume a very simple knowledge manager: the *knowledge items* are just sequences of values, i.e. *tuples*, and the operations on the knowledge manager permit adding, removing and reading tuples. Actions $\mathbf{get}(T)@c$ and $\mathbf{qry}(T)@c$ rely on *pattern-matching* wrt a given template (sequences of values and variables) to select a tuple from the tuple space and remove it or just take into account the necessary variable bindings to be used in the continuations[7]. Both these actions are blocking. A process executing $\mathbf{get}(T)@c$ or $\mathbf{qry}(T)@c$ is blocked until a tuple matching T is found. Action $\mathbf{put}(t)@c$ simply adds tuple t to the knowledge repositories of the components identified by c.

Composition of components and their interaction is implemented by exploiting the notion of *interface*. A component's interface can be inquired to extract information about the component, its status or its execution environment, as well as the services offered by the component. In fact, the interface provides a set of *attributes* characterising the component itself, which are simply names acting as references to information stored in the knowledge repository. For example, attributes might indicate the CPU load, the component's GPS position or, in addition, the provided services and their signature. Thus, components' composition and interaction rely on the attributes contained in the their interfaces. This form of semantics-based aggregation of components permits defining ensembles, which represent *social* or *technical networks* of autonomic components. The key point is that the formation rule is endogenous to components: components of an ensemble are connected by the interdependency relations defined through predicates over interfaces' attributes. In fact, an ensemble is not a rigid fixed network but rather a dynamic graph-like structure where component linkages are dynamically established. Therefore, no specific syntactic category or operator for forming ensembles is provided by SCEL, but they are dynamically 'synthesized' via group-oriented, attribute-based communication.

4.2 Robotics scenario in SCEL

In this section we show how the considered scenario can be modelled in SCEL. For the sake of simplicity we consider a simplified scenario where we have a single trapped robot[8]. Moreover, we consider only the part of the specification that shows how robots can reach the participant trapped in the hole. We assume that each robot is modelled via a SCEL node where the knowledge repository is *implemented* as a *tuple space*. Each robot is also equipped with *sensors* and *actuators*. Sensors can be used to collect data from the environment. Actuators are used to send commands to robot equipments (wheels, transmitters, gripper-based mechanism, etc.). In particular, we assume that each robot is equipped with the following sensors:

[7] A tuple matches a template if they have the same number of elements and corresponding elements have matching values or variables; variables match any value of the same type, and two values match only if they are identical.

[8] A detailed specification can be found at `http://code.google.com/p/jresp/`

- a *motion sensor* that can be used to verify if the robot is able or not to move;
- a *grasp sensor* that permits verifying if a robot can grasp another robot;
- a *GPS sensor* that identifies robot location.

Processes running at a robot can interact with the sensors outlined above by relying on standard operations on knowledge repository. This because we assume that sensors automatically publish read data in the knowledge repository. Knowledge items published by a sensor have the following structure:

$$\langle sensor_name, d_1, \ldots, d_n \rangle$$

where *sensor name* indicates the sensor that has published the item while d_1,..., d_n are the actual values read from the environment.

In our case an element of the form $\langle \mathsf{motion}, \mathsf{true} \rangle$ indicates that the robot can move (i.e. it is not trapped in a hole) while $\langle \mathsf{gps}, x, y \rangle$ states that robot is located at (x, y).

Each robot is also equipped with the following actuators:

- a *motion controller* that can be used to control wheels;
- a *rescue signal transmitter* that permits sending a rescue signal;
- a *gripper controller* that can be used to activate the gripper-based mechanism.

Like for sensors, processes interact with actuators by adding to the knowledge repository messages of the form:

$$\langle actuator_name, d_1, \ldots, d_n \rangle$$

where *actuator name* identifies the controlled actuator while d_1, ..., d_n are the data sent to the actuator. In our case we use messages of the form:

- $\langle \mathsf{wheels}, \delta \rangle$, to control wheels and rotate the robots towards angle δ;
- $\langle \mathsf{rescue}, b \rangle$, to start (when $b = \mathsf{true}$) or stop (when $b = \mathsf{false}$) emission of the rescue signal;
- $\langle \mathsf{gripper}, b \rangle$, to activate the gripper-based mechanism ($b = \mathsf{true}$ or $b = \mathsf{false}$ to activate or deactivate the mechanism).

Each robots publish in its status via the interface. Namely, we assume that $\mathcal{I}.\mathsf{rescue} = b$ if and only if a tuple of the form:

$$\langle \mathsf{rescue}, b \rangle$$

is in the robot local knowledge. If $\mathcal{I}.\mathsf{rescue} = true$ then the robot is trapped and it is emitting a rescue signal.

Controlling robot movements We assume that three processes run at each node: randomWalker, rescueReceiver and rescueHandler.

Process randomWalker is used to control the robot movements when no *rescue* signal has been received and when the robot is not trapped in a hole. Process code is the following:

```
randomWalker =def
    while ( true ) {
        qry(motion, !canMove)@self
        if (¬canMove) {
            put(rescue, true)@self.
            qry(motion, true)@self.
            put(rescue, false)@self.
        }
        qry(rescueMode, !isInvolved)@self
        if (isInvolved) {
            rescueHandler()
        } else {
            put(wheels, random(0, 2π))@self.
        }
    }
```

This process first checks if it the robot can move (i.e. if it is or not trapped). If the robot is trapped, a rescue signal is sent until the robot, thanks to the help of other robots, goes outside the hole. Process randomWalker also checks if the robot is involved in a mission for saving a trapped robot. Knowledge element $\langle \text{rescueMode}, b\rangle$ is used to notify the process if the current robot is involved or not in rescuing another participant (b is a boolean value). This tuple is always available in the robot knowledge repository. If the robot is *involved* then process rescueHandler() is executed. Otherwise, the robot moves towards a direction that is randomly selected in the interval $[0, 2\pi]$.

Process rescueReceiver is devoted to intercept rescue signals and set the robot direction towards the trapped robot:

```
rescueReceiver =def
    while ( true ) {
        qry(gps, !x, !y)@{I.rescue}
        put(wheels, towards(x, y)))@self.
        get(rescueMode, false))@self.
        put(rescueMode, true))@self.
        qry(rescueMode, false))@self.
    }
```

In the process above action $\mathbf{qry}(\text{gps}, !x, !y)@\{\mathcal{I}.\text{rescue}\}$ permits identifying the location (define in terms of gps-coordinates) of a trapped robot. This action is based on a *group* communication. Indeed, the target is predicate $\mathcal{I}$.rescue that is satisfied only by trapped robots.

As soon as the trapped robot is reached, process rescueHandler() activates the gripper-based mechanism and starts the activities (not specified here) to move the robot outside the hole:

```
rescueHandler =def
    qry(grasp, true)@self.
    put(grip, true)@self.
    Actions for rescuing the trapped robot
    put(grip, false)@self.
    get(rescueMode, true))@self.
    put(rescueMode, false))@self.
```

4.3 Running the scenario

To run the scenario considered in the previous section, we can rely on jRESP[9]. This is a runtime environment, developed in Java, that aims at providing programmers with a framework that permits developing autonomic and adaptive systems programmed in SCEL. A detailed description of jRESP can be found in [24, 8].

jRESP provides a set of API that permits using the SCEL paradigm in Java programs. This allows programmers to experiment with SCEL primitives that are integrated in a standard and well known programming language. For instance, the Java code programming the behaviour of random-Walker agent is reported in Figure 8.

```
while ( true ) {
    Tuple t = query(
        new Template(
            new ActualTemplateField("motion") ,
            new FormalTemplateField( Boolean.class )
        ) ,
        Self.SELF
    );
    boolean canMove = t.getElementAt( Boolean.class , 1 );
    if (!canMove) {
        put( new Tuple( "rescue" , true ) , Self.SELF );
        query(
            new Template(
                new ActualTemplateField("motion") ,
                new ActualTemplateField( true )
            ) ,
            Self.SELF
        );
    }
    t = query(
            new Template(
                new ActualTemplateField("rescueMode") ,
                new FormalTemplateField( Boolean.class )
            ) ,
            Self.SELF
        );
    boolean isInvolved = t.getElementAt( Boolean.class , 1 );
    if (isInvolved) {
        new rescueHandler().call();
    } else {
        put( new Tuple( "wheels" , Math.random()*Math.PI ) , Self.SELF );
    }
}
```

Figure 8: A portion of Java code using jRESP API

4.4 Soft Constraints as a linguistic abstraction in SCEL

Another possible approach to deal with autonomic computing issues, such as self-aware, self-adaptive and self-expressive, could be based on soft concurrent constraint programming, a programming paradigm with guard primitives like ask, tell and possibly guarantee negotiation constructs. A good example is cc-pi calculus [10]. However we are not interested in developing a new programming language, rather we think it is more productive to embed some of the design concepts and primitives of soft concurrent constraint programming into the ASCENS language SCEL. In fact, the linguistic constructs of SCEL

[9] http://code.google.com/p/jresp/

look quite convenient, equipped as they are with the logically defined notion of ensemble, for modeling both the dynamic interaction taking place during constraint propagation, and the structural nature of hierarchical constraints.

We mention two examples of nontrivial interaction between the procedural and the declarative aspects of autonomic components. One is about emerging knowledge, described in Section 3.2.3: in the ordinary execution phase the behavior is guided by the present knowledge of the robot, which however tends to become obsolete due to “cyberphysical” errors. In the constraint propagation phase, the present knowledge is procedurally updated on the basis of deduction procedures activated by the interaction with other robots, in general we could say by additional interaction with the environment. When seen in SCEL terms, the constraint propagation phase could be interpreted as a generalized multiparty transactional synchronization step, where the ensemble coordinator asks for everybody knowledge, derives all the additional consequences, and returns them to the ensemble components.

The second example is about constraints for service contracts, in the style of [9], presented in ASCENS deliverable D2.1. The participants negotiate their behaviours, and if an agreement is reached they commit and start an execution which is guaranteed to be stuck-free. More precisely, the first phase consists of a compilation step which generates for every client and every server separately a constraint modelling its behaviour and an instrumented compiled code. The second step is simply constraint composition for the client and the server which want to interact: it is successful (i.e. the result is consistent) only if the resulting constraint is satisfiable. Finally, the actual execution is monitored by ask-like guards present in the instrumentation, which forbid interactions leading to a stuck situation. Here the advantage of a constraint based approach is clear: the necessary constraints can be built inductively at compile time, composed at matching time and tested at run time taking advantage of concepts well-studied in the area of constraint programming.

We plan to investigate if the instrumented, compiled code could be programmed in a suitable SCEL dialect. Also a compilation tool could be considered.

5 Conclusions

We have briefly introduced the main formalism developed in ASCENS to support development of autonomic components during the different stages of the software cycle. Moreover we have given an indication of how they can be exploited by showing how they are can be used to model components inspired by a simple scenario dealing with robot swarms operating in an open environments within which they have to collaborate to achieve assigned goals.

This work has to be considered as a first step in the direction of understanding the connections between the different formalism that are developed, and used, by very different communities. Indeed, what we plan for next year is to move from a common scenario to a common case study. This will still be based on one of the three scenarios considered in the project but, being more concrete and detailed will allow us to evaluate how abstract specifications providing different level of details in SOTA and GEM/POEM can be used as the starting points to develop a running piece of Java code that is built directly from the SCEL model. We will also experiment with the use of different knowledge representation mechanisms such as those based on KnowLang and/or on concurrent constraint. For the latter, we will continue our experiments with adding constraint-based repository to SCEL to assess the gain with respect to the tuple based approach. Again, in the direction of integration with research pursued in different work packages we will investigate how properties of SCEL programs can be guaranteed by exploiting the operational semantics of the language and by mapping the transition system associated to a program into the internal representation of set of BIP-based verification tools that are heavily used in WP5.

References

[1] D. B. Abeywickrama, N. Bicocchi, and F. Zambonelli. Sota: Towards a general model for self-adaptive systems. In *21st IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2012, Toulouse, France, June 25-27, 2012*, pages 48–53, 2012.

[2] K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.

[3] S. Bensalem, M. Boreale, M. Loreti, R. Bruni, A. Corradini, F. Gadducci, U. Montanari, M. Sammartino, M. G. Buscemi, R. De Nicola, A. Lluch Lafuente, A. Vandin, G. Cabri, D. Latella, and M. Massink. D2.1: First report on wp2. enhanced connectors, resource-aware operational models and the negotiate-commit-execute schema and its foundations, 2011. ASCENS Deliverable.

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

[5] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based contstraint logic programming: syntax and semantics. *ACM Trans. Program. Lang. Syst.*, 23(1):1–29, 2001.

[6] S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Log.*, 7(3):563–589, 2006.

[7] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. A conceptual framework for adaptation. In J. de Lara and A. Zisman, editors, *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering, FASE 2012*, volume 7212 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2012.

[8] T. Bureš, V. Horký, J. Keznikl, J. Kofroň, M. Loreti, and F. Plášil. Language extensions for implementation-level conformance checking. ASCENS Deliverable D1.5, September 2012.

[9] M. G. Buscemi, M. Coppo, M. Dezani-Ciancaglini, and U. Montanari. Constraints for service contracts. In R. Bruni and V. Sassone, editors, *TGC*, volume 7173 of *Lecture Notes in Computer Science*, pages 104–120. Springer, 2011.

[10] M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In R. De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2007.

[11] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. Languages primitives for coordination, resource negotiation, and task description. ASCENS Deliverable D1.1, September 2011. `http://rap.dsi.unifi.it/scel/`.

[12] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A language-based approach to autonomic computing. In *Proc. of the 10th International Symposium on Software Technologies Concertation on Formal Methods for Components and Objects (FMCO 2011)*, LNCS 7542, pages 25–48. Springer, 2012. `http://rap.dsi.unifi.it/scel/`.

[13] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. SCEL: a Language for Autonomic Computing. Technical Report, September 2012. `http://rap.dsi.unifi.it/scel/`.

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

[15] A. Gutiérrez, A. Campo, F. C. Santos, C. Pinciroli, and M. Dorigo. Social odometry in populations of autonomous robots. In *Proceedings of the 6th international conference on Ant Colony Optimization and Swarm Intelligence*, ANTS '08, pages 371–378, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] M. G. Hinchey and R. Sterritt. Self-managing software. *IEEE Computer*, 39(2):107–109, 2006.

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

[18] M. Hölzl, L. Belzner, A. Klarl, and C. Kroiss. D8.2: Second report on wp8: The ascens service component repository (first version).

[19] M. M. Hölzl and M. Wirsing. Towards a system model for ensembles. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 241–261. Springer, 2011.

[20] IBM Corporation. *An Architectural Blueprint for Autonomic Computing*, 2006.

[21] S. Kounev. Self-Aware Software and Systems Engineering: A Vision and Research Roadmap. In *GI Softwaretechnik-Trends, 31(4), November 2011, ISSN 0720-8928*, Karlsruhe, Germany, 2011.

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

[23] T. Lints. The essentials in defining adaptation. In *Proceedings of the 4th Annual IEEE Systems Conference*, pages 113–116, 2010.

[24] M. Loreti. jresp: a run-time environment for scel programs. Technical Report, September 2012. `http://rap.dsi.unifi.it/scel/`.

[25] R. Milner. Calculi for interaction. *Acta Inf.*, 33(8):707–737, 1996.

[26] F. Mondada, G. C. Pettinaro, A. Guignard, I. W. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. M. Gambardella, and M. Dorigo. Swarm-bot: A new distributed robotic concept. *Autonomous Robots*, 17(2-3):193–221, 2004.

[27] G. V. Monreale and U. Montanari. Soft constraint logic programming for electric vehicle travel optimization. In *WLP 2012*, 2012.

[28] U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.

[29] U. Montanari and E. Vassev. Soft constraints for knowlang. In *FMSAS 2012*. ACM Press, 2012.

[30] M. Morandini, L. Sabatucci, A. Siena, J. Mylopoulos, L. Penserini, A. Perini, and A. Susi. On the use of the goal-oriented paradigm for system design and law compliance reasoning. In *iStar 2010–Proceedings of the 4 th International i* Workshop*, page 71, Hammamet, Tunisia, June 2010.

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

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

[33] R. O'Grady, R. Groß, A. L. Christensen, and M. Dorigo. Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots*, 28(4):439–455, 2010.

[34] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications*, 14(3), 1999.

[35] R. Pugliese, T. Bures, R. D. Nicola, J. Keznikl, M. Loreti, F. Plasil, and F. Tiezzi. D1.2: Second Report on WP1 Languages for Coordinating Ensemble Components, 2012. ASCENS Deliverable, D1.2, October 2012.

[36] O. Pustovalova and U. Montanari. Constraint Logic Programming for Service-Oriented Computing: A Case Study in Prova. Technical report, IMT Institute for Advanced Studies Lucca, 2012. Available online at `http://www.imtlucca.it/olga.pustovalova`.

[37] K. Rasch, F. Li, S. Sehic, R. Ayani, and S. Dustdar. Context-driven personalized service discovery in pervasive environments. *World Wide Web*, 14(4):295–319, 2011.

[38] P. Robertson, H. E. Shrobe, and R. Laddaga, editors. *Self-Adaptive Software, First International Workshop, IWSAS 2000, Oxford, UK, April 17-19, 2000, Revised Papers*, volume 1936 of *LNCS*. Springer, 2001.

[39] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), 2009.

[40] M. Thielscher. *Action Programming Languages*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2008.

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

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

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

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

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

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

[47] E. Vassev, M. Hinchey, B. Gaudin, P. Nixon, N. Bicocchi, and F. Zambonelli. D3.1: First Report on WP3. Knowledge Representation for Self-Awareness, 2011. ASCENS Deliverable.

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

[49] N. Šerbedžija, M. Massink, M. Brambilla, D. Latella, M. Dorigo, M. Birattari, N. Hoch, H. P. Bensler, D. Abeywickrama, J. Keznikl, I. Gerostathopoulos, and T. Bures. D7.2: Second Report on WP7. Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility, 2012. ASCENS Deliverable.

[50] N. Šerbedžija, S. Reiter, M. Ahrens, J. Velasco, C. Pinciroli, N. Hoch, and B. Werther. D7.1: First Report on WP7. Requirement specication and Scenario description, 2012. ASCENS Deliverable.

[51] F. Zambonelli, D. B. Abeywickrama, N. Bicocchi, M. Puviani, R. Pugliese, and E. Vassev. D4.1.0: First Report on WP4, 2011. ASCENS Deliverable.

[52] F. Zambonelli, N. Bicocchi, G. Cabri, L. Leonardi, and M. Puviani. On self-adaptation, self-expression, and self-awareness, in autonomic service component ensembles. In *Proceedings of the AWARENESS Workshop at the 5th IEEE International Conference on Self-adaptive and Self-organizing Systems*, 2011.